UNIVERSITÉ DE NANTES

FACULTÉ DES SCIENCES ET DES TECHNIQUES

———

ÉCOLE DOCTORALE STIM (SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET MATHÉMATIQUES)

N° attribué par la bibliothèque

Année 2014

Contribution à l'automatisation et à l'évaluation des architectures logicielles ouvertes

———

THÈSE DE DOCTORAT

Discipline : Informatique

Spécialité : Génie logiciel

Présentée
et soutenue publiquement par

Riad BELKHATIR

Le 18 Juin 2014 à l'UFR Sciences et Techniques, Université de Nantes,
devant le jury ci-dessous

Présidente	Isabelle BORNE, Professeur	Université de Bretagne-Sud
Rapporteurs	Nicole LÉVY, Professeur	Conservatoire National des Arts et Métiers
	Pascal POIZAT, Professeur	Université Paris Ouest Nanterre la Défense
Examinateurs	Isabelle BORNE, Professeur	Université de Bretagne-Sud
	Christian ATTIOGBÉ, Professeur	Université de Nantes
	Arnaud VIGUIER, Directeur	BeOtic

Directeur de thèse : Professeur Mourad OUSSALAH

ED : STIM

CONTRIBUTION À L'AUTOMATISATION ET À L'EVALUATION DES ARCHITECTURES LOGICIELLES OUVERTES

Contribution to the automation and to the evaluation of open-system software architectures

Riad BELKHATIR

Sigillum civitatis Gratianopolis

Université de Nantes

Riad BELKHATIR

Contribution à l'automatisation et à l'évaluation des architectures logicielles ouvertes

193 pages

A mon père et ma mère,
pour m'avoir toujours envoyé chercher,
ce qu'ils auraient pu se contenter de me donner.

Remerciements

Cette première page est l'occasion d'exprimer mes sincères remerciements à celles et ceux qui ont, au fil des années, joué un rôle dans la réalisation de ce travail.
Mes remerciements s'adressent d'abord à l'ensemble des membres de mon jury.
Je commencerai par remercier Isabelle Borne, professeur à l'Université de Bretagne-Sud pour avoir accepté de présider mon jury de thèse.
Je suis très honoré que Nicole Lévy, professeur au Conservatoire National des Arts et Métiers et Pascal Poizat, professeur à l'Université Paris Ouest Nanterre la Défense, aient accepté d'être rapporteurs de ma thèse. Je remercie également Christian Attiogbé, professeur à l'Université de Nantes d'avoir bien voulu participer à ce jury.
Je remercie chaleureusement mon directeur de thèse, Mourad Oussalah, professeur à l'Université de Nantes, pour son aide et les précieux conseils qu'il m'a dispensés tout au long de cette thèse ; je le remercie pour les relectures souvent tardives et autres petits tracas et lui transmets toute ma reconnaissance. Je remercie également Arnaud Viguier et Cédric Couton pour avoir cru en moi, pour m'avoir accueilli au sein de leur équipe à BeOtic et offert les moyens de mener ce travail à bien. Les discussions que nous avons eues m'ont permis d'avancer dans cette thèse et dans ma perception de notre travail.

Je tiens à remercier l'ensemble du personnel du LINA pour leur patience et leur aide dans les méandres de l'administration. Je pense également à mes collègues de BeOtic qui m'ont formidablement bien intégré, ce qui a fortement contribué à la qualité du travail fourni. Je pense également aux enseignants que j'ai eus depuis le primaire. J'ai rencontré parmi eux beaucoup de gens passionnés et passionnant qui m'ont encouragé. Ce qu'ils m'ont appris m'a donné l'envie de savoir, de comprendre et de chercher. Ils m'ont donné l'envie de faire ce que je fais aujourd'hui.

Enfin, je remercie bien évidemment ma famille. Je dois un merci particulier à mes parents. Merci pour m'avoir conseillé, accompagné, réprimandé etc.. Ce manuscrit ne serait pas le même sans vos précieuses recommandations. Vos encouragements et votre soutien ont été ma lumière d'Eärendil pendant toute la rédaction.
Merci Mohammed et Hind pour votre aide morale et financière ; vos encouragements, et vos présences m'ont conduit ici. Je n'aurais jamais pu réaliser tout ça sans votre soutien sans faille. Ca y est, j'y suis, à mon tour.

Table des matières

Table des figures

Liste des tableaux

Introduction

Cadre de la thèse

Cette thèse CIFRE[1] s'inscrit dans le cadre de l'ingénierie logicielle et se concentre essentiellement sur les procédés d'automatisation et d'évaluation de la qualité des architectures logicielles ouvertes et plus précisément sur les paradigmes architecturaux orientés services. Elle a été réalisée entre le laboratoire LINA[2] de l'Université de Nantes, dans l'équipe AeLoS[3] spécialisée dans les architectures logicielles et l'entreprise nantaise BeOtic[4], spécialisée dans l'édition de logiciels innovants, et ayant pour mission d'accompagner les entreprises dans l'industrialisation de leurs processus projets (pilotage de projet, automatisation des processus et amélioration continue de la qualité).

[1] Conventions Industrielles de Formation par la REcherche
[2] Laboratoire Informatique de Nantes Atlantique
[3] Architectures et Logiciels Sûrs
[4] http://www.beotic.fr

Les paradigmes architecturaux sont des patrons de conception pour la structure et l'interconnexion entre les systèmes logiciels [ACK+02] et leur évolution est généralement liée à l'évolution de la technologie.
Un paradigme architectural définit des groupes de systèmes en termes de :

- modèles de structures
- vocabulaires de composants et de connecteurs
- règles ou contraintes sur des relations entre systèmes [GS94]

Nous pouvons distinguer plusieurs paradigmes architecturaux pour les systèmes distribués, et, parmi les plus notoires, trois ont fortement contribué à l'évolution des problématiques. Il s'agit chronologiquement des paradigmes orientés objet (OOA[5]), des paradigmes orientés composant (CBA[6]) et de ceux orientés service (SOA[7]).

Les premiers développeurs se rendirent rapidement compte des répétitions de code dans les applications et cherchèrent ainsi à définir des mécanismes limitant ces répétitions. C'est ainsi qu'apparurent les architectures orientées objet (OOA), qui fournissent un contrôle amélioré de la **réutilisabilité** (*capacité à pouvoir réutiliser un système de la même manière ou après un certain nombre de modifications*). Cette approche permit d'ouvrir la voie à des applications de plus en plus complexes et par conséquent à l'identification de nouvelles limites en termes de granularité.

Ces limites ont mené à la recentralisation des problématiques vers la **composabilité** (*capacité à pouvoir combiner d'une manière sûre les éléments architecturaux afin d'établir de nouveaux systèmes ou éléments architecturaux composés*).

Corrélativement, la communauté de l'ingénierie logicielle a développé et présenté CBA pour surmonter ce nouveau défi. Ainsi, le CBA renforce le contrôle de la composabilité et formalise clairement les processus associés. Par extension, cette

[5] Object Oriented Architecture
[6] Component Based Architecture
[7] Service Oriented Architecture

formalisation établit la base nécessaire aux possibilités d'automatisation. Dans le même temps, une partie de la communauté du logiciel a entamé des recherches dans une nouvelle direction : la **dynamicité** (*capacité à développer des applications capables d'adapter leur comportement de manières dynamique, automatique et autonome afin de répondre aux besoins liés aux changements des exigences et des contextes ainsi qu'aux possibilités de défaillances*). C'est ainsi que les architectures orientées service (SOA) ont été développées sur la base de l'expérience acquise par les objets et les composants, avec une focalisation majeure sur la manière d'améliorer la dynamicité.

Dans le cadre de la thèse, il nous a donc semblé naturel de nous focaliser exclusivement sur la technologie SOA car c'est la plus récente et la plus complète au regard des deux autres technologies, sur lesquelles se basent d'ailleurs ses fondements. De plus, l'entreprise BeOtic avec qui nous avons effectué cette thèse utilise majoritairement la technologie SOA pour implémenter ses solutions dans les domaines des plateformes collaboratives et de la gestion de projets, ce qui nous a confortés dans notre choix.

L'architecture orientée service (SOA) est un paradigme architectural populaire, ayant pour but de modéliser et de concevoir des systèmes distribués [Kim08].

Les solutions SOA ont été créées pour satisfaire les objectifs commerciaux tels que la création de nouveaux services pour les utilisateurs, la baisse des coûts tout en restant compétitifs et l'intégration de la nouvelle solution aux systèmes existants. Ces dernières années, l'architecture orientée service (SOA) a connu une ascension croissante et de plus en plus de sociétés ont été séduites par cette technologie et ses forces (réutilisabilité, baisse des coûts et augmentation de la productivité), notamment en raison d'un contrôle amélioré des attentes liées à l'aspect business. Cette technologie peut en somme apporter beaucoup d'avantages mais elle peut également engendrer des complications importantes perturbant ainsi l'organisation générale de la société qui tente de l'adopter.

Dans cette perspective, afin de rendre un système robuste, il est nécessaire que son architecture puisse répondre favorablement aux exigences dites fonctionnelles (ce

qu'un système est supposé faire ; ceci définit les fonctions du système) et aux exigences dites non fonctionnelles ("ce qu'un système est supposé être", autrement dit sa qualité) [LMB07].

Rendre un système robuste n'est pas chose aisée car développer des SOA peut engendrer de nombreux risques tant la complexité de cette technologie est notable, en particulier pour l'orchestration de services [Maz06] et également en raison de la nature complexe des problématiques financières que cette technologie engendre. De plus, les progressions en termes de taille du logiciel rendent le développement plus complexe à manipuler, et celles-ci font que toute forme de prévisibilité (coût et qualité) devient extrêmement difficile. Justement, de larges projets ayant coutés plusieurs millions de dollars et lancés par des entreprises majeures (telles que Ford ou GSA[8]) [Swe06] ont totalement échoué et ont donc été abandonnés car l'intégration de la nouvelle solution orientée service dans le système existant a été effectuée de manière désordonnée.

La qualité d'un logiciel étant l'un des enjeux majeurs dans la conception des outils logiciels, il existe en premier lieu, parmi les risques évoqués, celui de ne pas pouvoir répondre favorablement aux attentes en termes de qualité de services car les qualités clés de l'architecture que l'on appelle communément attributs qualité dérivent automatiquement des exigences commerciales.

Puisque ces risques peuvent potentiellement être distribués à tous les services, la question d'évaluer la cohérence du projet ainsi que la qualité de l'architecture choisie et donc chacun des attributs qualité des SOA s'est donc récemment posée. Ceci permet essentiellement de [BKM07]:

(i) Contrôler les différents coûts (en termes de temps et d'argent).
(ii) Apporter beaucoup plus de crédibilité au projet.
(iii) Se distinguer de la concurrence.
(iv) Mener aux certifications (normes).

[8] General Services Administration

(v) Identifier et corriger les erreurs persévérantes qui pourraient s'être révélées après l'étape de conception du logiciel.
(vi) Éviter tout futur danger significatif, y compris les abandons de projet qu'une telle évolution pourrait potentiellement entraîner.

L'analyse et l'évaluation de la qualité des logiciels sont des phases de la vie du logiciel ayant connu un essor certain dans la communauté des systèmes informatiques durant ces trois dernières décennies au point d'être devenues de véritables étapes cruciales. Afin d'évaluer la qualité d'un système par rapport aux exigences de ses clients, les concepteurs ont besoin de méthodes et d'outils d'analyse durant la phase d'évaluation. L'objectif primordial de l'évaluation de l'architecture d'un logiciel est de l'analyser afin d'identifier les risques potentiels et de vérifier que les exigences qualité ont été abordées dans la phase de conception.

Problématique abordée

De plus en plus de sociétés choisissent les solutions SOA pour développer leur architecture et comme nous l'évoquions précédemment, l'évaluation des SOA prend de plus en plus d'ampleur puisque c'est le pont entre le système et les objectifs commerciaux de l'organisation dans la mesure où l'évaluation permet de noter les différents attributs qualité (dérivant directement des objectifs commerciaux) des services composant le système [LMB07].

L'évaluation de l'architecture orientée service se réfère :

- Aux approches qualitatives et quantitatives.
- À la prévision de la charge associée aux évolutions.
- Aux limites théoriques d'une architecture donnée.

Dans cette perspective, les outils et les approches existants ont montré leurs limites pour l'évaluation de la qualité des SOA [CKK01] car aucun d'entre eux n'a clairement pu démontrer son efficacité dans la mesure où le travail d'évaluation réalisé est beaucoup trop abstrait car non automatisable, et ce problème est retranscrit dans la nature des fruits de l'évaluation qui ne délivrent aucun résultat précis mais des scénarios de tests à exécuter pour s'assurer que l'architecture est la bonne pour l'organisation. C'est ainsi que l'ampleur de la tâche a incité le monde académique à se positionner sur ces problématiques connues afin d'essayer de développer une approche plus formelle et générique que celle des méthodes existantes (ATAM[9], SAAM[10]) pour évaluer les SOA.

Notre travail de recherche s'inscrit dans cette même problématique. Après avoir étudié les méthodes existantes, nous proposons une approche différente pour évaluer la qualité des architectures orientées service.

Nous tentons en effet de fournir des résultats d'évaluation de la qualité des SOA en essayant de restreindre au maximum l'intervention humaine lors de l'évaluation afin d'automatiser le processus et d'obtenir des résultats quantifiables.

Contributions

Les contributions de cette thèse s'organisent autour des trois axes de recherche suivants :

-la définition d'un modèle prévisionnel de qualité. Ce modèle de décomposition de l'architecture sera organisé autour de plusieurs attributs qualité.

-la conception d'une méthode d'évaluation de la qualité d'une SOA basée sur le modèle prévisionnel de qualité.

---

[9] Architecture tradeoff analysis method

[10] Software architecture analysis method

-l'implémentation d'un prototype permettant une telle évaluation et retournant les résultats attendus.

La première contribution est la spécification d'un modèle prévisionnel de qualité, appelé modèle SOAQE (pour *Service Oriented Architecture Quality Evaluation*), capable de décomposer toute architecture orientée service en plusieurs attributs qualité. Nous avons répertorié ces attributs qualité en quatre catégories distinctes :

- les points de vue qualité.
- les facteurs qualité.
- les critères qualité.
- les métriques qualité.

Pour concevoir ce modèle de qualité, nous nous sommes inspirés du modèle de McCall imaginé en 1977 [MRW77]. Ce modèle décrivant la qualité logicielle et qui mena à la norme internationale pour l'évaluation de la qualité logicielle ISO/IEC 9126-1:2001 [CF06] (puis dernièrement ISO/IEC 25010:2011 [SAA03]) décompose également les attributs qualité, mais en trois étapes seulement. Nous avons identifié un niveau d'abstraction supérieur supplémentaire (les points de vue qualité) car il nous semblait judicieux de séparer les problématiques liées à la qualité logicielle en familles d'attributs qualité de hauts niveaux (par exemple point de vue business, et point de vue architectural).

La seconde contribution concerne la conception de la méthode SOAQE permettant l'évaluation de la qualité d'une SOA. Cette méthode qui est entièrement basée sur le modèle que nous avons conçu permet d'évaluer les SOA en combinant une approche automatisée puis l'intervention humaine afin d'éliminer les défaillances identifiées avec les méthodes de ces dernières années telles qu'ATAM, SAAM ou ARID[11] [CKK01]. Cette méthode SOAQE, du fait de sa nature semi automatisée, corrige les défaillances observées jusque là tels que le manque de pertinence et

[11] Active Reviews for Intermediate Designs

d'exactitude des résultats. Le processus consiste en quatre étapes principales correspondantes aux quatre niveaux de décomposition de notre architecture ; en effet, notre méthode permet à toute architecture SOA soumise, d'être systématiquement décomposée en points de vue qualité, chacun d'entre eux sont découpés en plusieurs facteurs qualités, caractérisés par plusieurs critères qualité étant eux-mêmes composés par différentes métriques qualité (chacune étant précisément quantifiable). Dans cette arborescence, chacun des attributs qualité en question est pondéré de manière différente en fonction du point de vue concerné (intervention humaine, d'où la nature semi-automatisée et non simplement automatisée de notre méthode). Ainsi, il est possible, à partir de la valeur de chacune des métriques, jusqu'à l'ensemble des points de vues qualité composant l'architecture SOA, de déterminer une note générique pour la qualité de l'architecture par le moyen d'agrégations et de combinaisons de valeurs des attributs qualité.

La troisième contribution expérimente le travail élaboré pendant notre thèse car celle-ci concerne l'implémentation d'un prototype fonctionnel réalisé autour du modèle et de la méthode SOAQE que nous avons développés. L'outil que nous avons réalisé permet d'évaluer la qualité d'une architecture orientée service reçue en entrée et retourne une combinaison de résultats sous formes textuelle et graphique pour une meilleure compréhension des rapports d'évaluation, et facilitant ainsi, une éventuelle correction.

La figure 0.1 résume ces contributions suivant les trois axes de recherche et leurs articulations les uns par rapport aux autres.

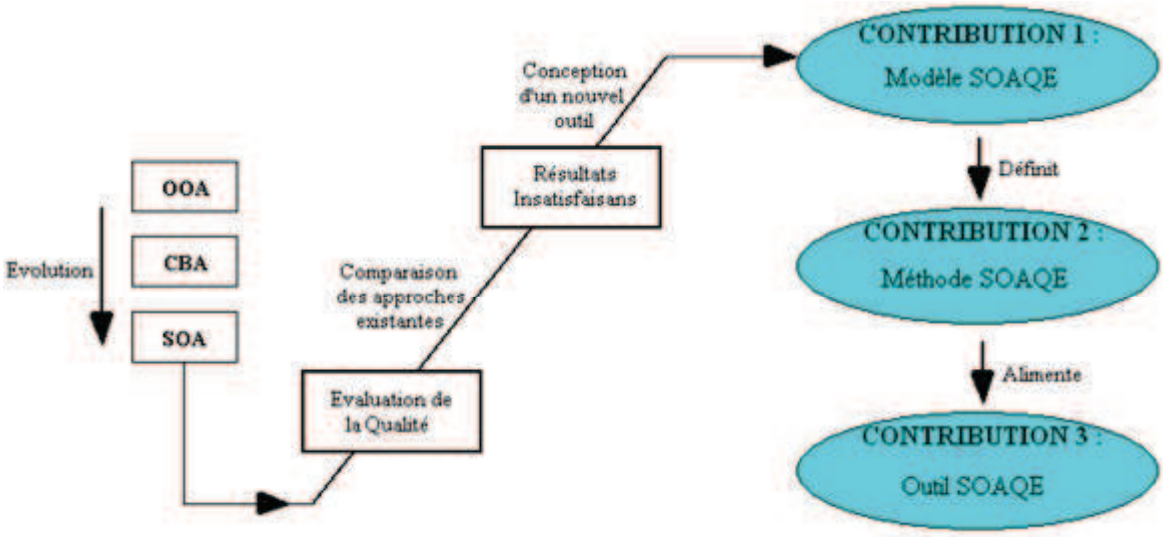

Fig. 0-1 - Organisation du travail de recherche et des contributions

Organisation du document

Ce manuscrit de thèse est composé de quatre chapitres :

Le chapitre 1 pose le socle bibliographique de notre travail. Dans un premier temps, il fait état du domaine des architectures logicielles et nous y présentons essentiellement sa terminologie et ses concepts. Dans un second temps, nous présentons les divers paradigmes architecturaux incontournables en introduisant leurs définitions et concepts et en expliquant par quels procédés ils ont pu marquer leur temps. Nous continuons ce chapitre en expliquant l'intérêt de se concentrer exclusivement sur les architectures orientées service en revenant sur leur origine et les motivations ayant poussé leur développement. Ensuite, nous tentons de donner une définition claire de la qualité logicielle à travers les définitions d'auteurs et de chercheurs notoires puis nous tenterons d'expliquer pourquoi il s'agit aujourd'hui d'une préoccupation croissante et quels sont les intérêts de mesurer la qualité d'une architecture orientée service. Nous établissons ensuite une rétrospective des modèles de qualité les plus connus sous forme d'une liste non exhaustive, nous y présentons leurs définitions et leurs concepts et nous expliquons comment chacun d'entre eux a

participé à l'évolution des problématiques. Finalement, nous présentons dans ce chapitre les attributs qualité clés, communs à plusieurs modèles qualité. Nous nous attarderons logiquement davantage sur le modèle de McCall [MRW77] et sa déclinaison, le modèle ISO/IEC 9126-1 [CF06] ayant tous deux directement servis de bases pour la réalisation de l'ensemble de nos travaux.

Le chapitre 2 présente le concept de l'évaluation de l'architecture. Dans un premier temps, nous introduisons ce nouveau chapitre en y expliquant quels sont les intérêts majeurs de l'évaluation de la qualité d'une SOA et nous poursuivons en dressant une liste des principales méthodes d'évaluation ayant fait leur apparition dans le domaine du génie logiciel ces dernières décennies. Nous expliquerons leurs forces et leurs failles et tenterons d'expliquer en analysant le résultat de leurs évaluations respectives, pourquoi il n'existe toujours pas de solution satisfaisante quant à l'obtention de résultats d'évaluation précis, clairs et quantifiables (sous forme de notes ou de pourcentages). Dans un second temps, nous introduirons quelques outils d'évaluation notoires, leurs fonctionnalités et la précision de leurs résultats et de même que pour les méthodes, nous détaillerons comment notre outil pourra répondre aux manquements observés avec les outils existants.

Le chapitre 3 met en évidence le modèle SOAQE et détaille chacune des étapes de la méthode éponyme. D'abord nous expliquerons pourquoi nous avons choisi de décomposer les SOA de cette façon tout en précisant pourquoi nous avons choisi de travailler avec des attributs qualité précis et récurrents que nous définirons en détails. Nous reviendrons également sur les choix des coefficients de pondération des attributs qualité en fonction du point de vue étudié.

Le chapitre 4 décrit quant à lui l'implémentation d'un prototype dérivant du modèle et de la méthode SOAQE. Nous y décrirons son aspect "*réalisationnel*", à travers une analyse de son architecture technique et des étapes nous ayant menés à sa réalisation, tels que les choix des différentes technologies utilisées lors de l'implémentation. Nous y décrirons également ses différentes fonctions à travers un projet existant à BeOtic nous permettant d'illustrer l'outil SOAQE.

En guise de conclusion, nous faisons le bilan précis de ce travail de thèse en soulignant nos participations principales et en mettant en valeur leurs apports dans le domaine des architectures orientées service. Aussi, nous dressons un bilan critique sur la recherche et les travaux effectués pendant ces trois années et nous suggérons un ensemble de pistes d'extensions possibles. La structure générique du manuscrit de thèse est présentée dans la figure 0.2.

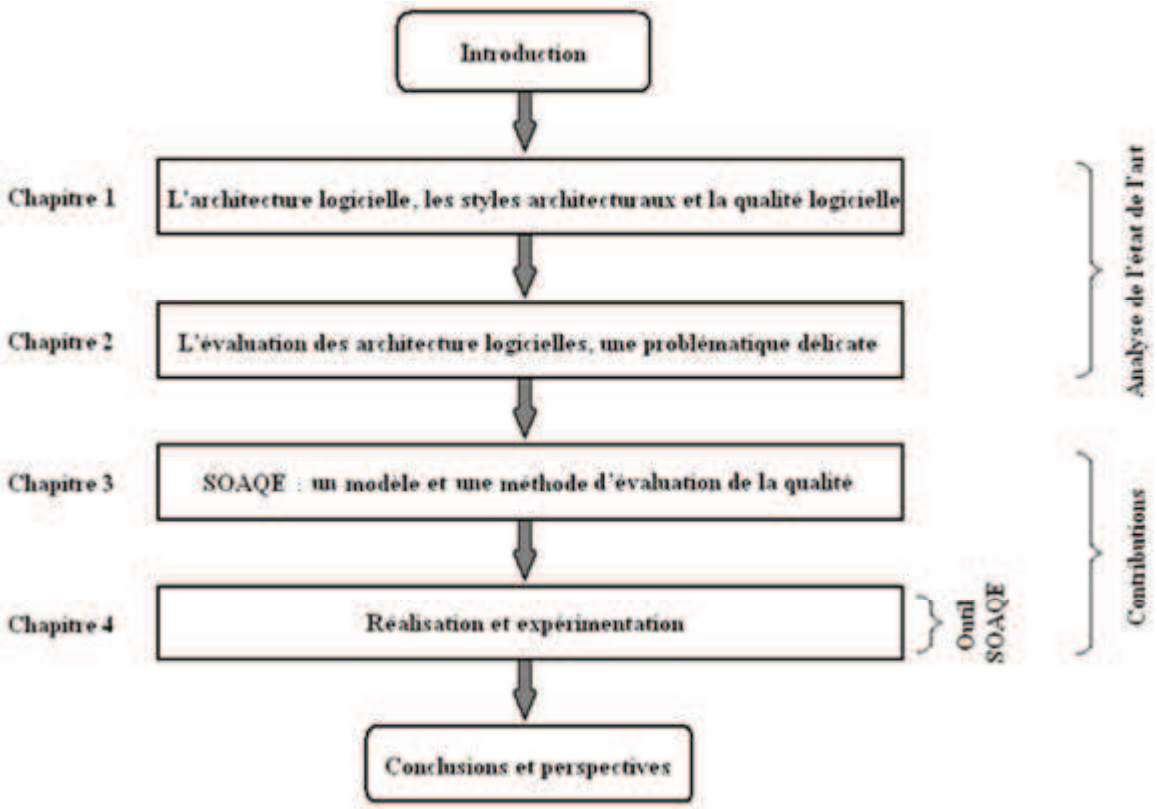

Fig. 0-2 - Représentation schématique de la structure globale de la thèse

Chapitre 1

L'ARCHITECTURE LOGICIELLE, LES STYLES ARCHITECTURAUX ET LA QUALITÉ LOGICIELLE

1.1 Introduction

Garlan et Perry [PW92, SG96] ont décrit l'architecture logicielle comme étant une "structure de composants dans un système, les relations entre ces composants, les principes et les directives qui contrôlent la conception et l'évolution dans le temps".

Bass [BCK03], quant à lui, met l'accent uniquement sur les aspects internes d'un système et définit l'architecture logicielle d'un système comme étant la structure ou les structures du système.

Ces structures comprennent les composants logiciels, les propriétés visibles de ces composants, et les relations entre eux. [BCK03]

Par propriétés "visibles", nous entendons par là ses services fournis, ses caractéristiques de performance, sa gestion des défaillances, l'utilisation de ressources partagées, etc.

L'idée de cette définition consiste à dire qu'une architecture logicielle doit abstraire quelques informations du système (il n'y aurait sinon aucun intérêt à étudier l'architecture, nous ne serions simplement qu'en train de visualiser le système dans sa globalité) et doit fournir assez d'informations pour être une base solide pour l'analyse, la prise de décision, et par conséquent la réduction des risques.

L'absence d'une architecture pour le logiciel le mène directement à sa perte comme nous pouvons le voir sur la figure 1-1. En effet, nous y voyons clairement que la mise en place d'une architecture permet d'éradiquer continuellement les défaillances initialement observées ; tandis que lorsque nous n'en n'établissons aucune, il existe un risque important, notamment lorsque l'on opère des changements dans le système. C'est à ce moment précis qu'une augmentation brutale du taux de défaillance fait son apparition du fait des effets de bord.

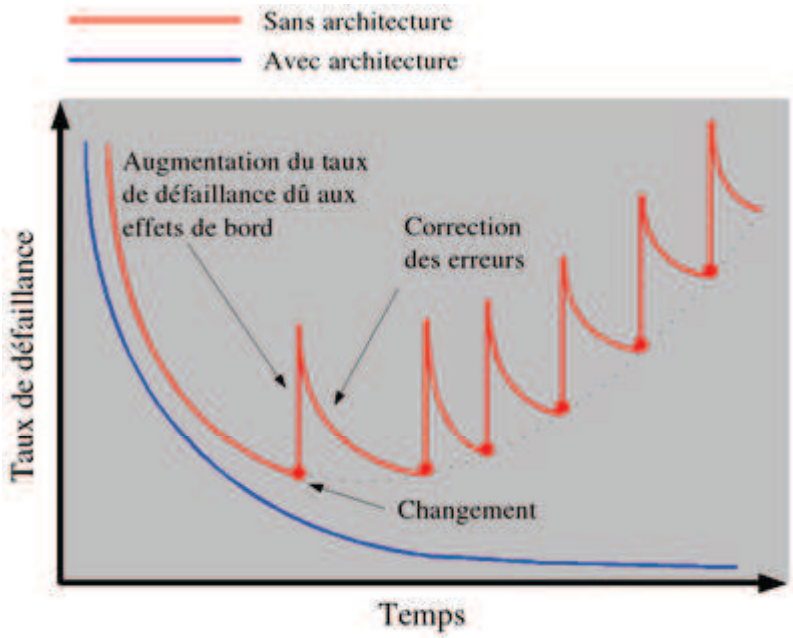

Fig. 1-1 - Graphique de dégradation du logiciel en fonction de l'architecture.

L'architecture définit les composants (tels que les modules, les objets, les processus, les sous-systèmes, les unités de compilation, et ainsi de suite) [BDH+98, GKT05] et les relations (telles que les appels, les échanges de données, les synchronisations, les utilisations, les dépendances, les instanciations, et encore plus) entre eux [GHJ+94, Szy02, Jon11].

L'architecture est le résultat des premières décisions de conception qui sont nécessaires avant qu'un groupe d'intervenants collaborant puisse construire un système logiciel. [SG96, GS94]

Plus ce groupe est consistant, plus l'architecture est vitale (le groupe n'a cependant pas à être très large pour que l'architecture devienne vitale).

1.2 Que peut-on qualifier d'architectural ?

Les intervenants en charge de l'évaluation logicielle ont un besoin majeur de définir précisément ce qu'est l'architecture et quelles informations sont à considérer dans

leur domaine de compétences. Les questions que l'on se pose habituellement à ce sujet sont [Fab07]:

- Quelle est la différence entre une architecture et une conception de haut niveau ?
- Est-ce que des détails tels que les priorités des processus sont architecturaux ?
- Pourquoi les questions d'implémentation sont-elles traitées comme étant architecturales ?
- Les interfaces des composants sont-elles une partie de l'architecture ?
- L'architecture est-elle concernée par le comportement lors de l'exécution ou la structure statique ?
- La partie "système d'exploitation" ou encore le langage de programmation font-ils partie de l'architecture ?

Pour tenter de répondre à toutes ces questions, nous pouvons d'abord commencer par rappeler la définition de Bass que nous avons précédemment citée : "une architecture logicielle décrit un système en termes de composants, de leurs propriétés visibles, et leurs relations entre eux". Cette définition ne considère pas explicitement la notion du contexte et reste assez floue. Il a été considéré que les algorithmes, les structures de données et les détails du flux de données n'étaient pas architecturaux [BCK03].

Ces déclarations ne sont que partiellement vraies. Quelques propriétés des algorithmes, telles que leur complexité par exemple, pourraient avoir un effet dramatique sur la performance. Quelques propriétés des structures de données peuvent également affecter directement la performance et la fiabilité. Certains détails du flux de données impactent également la *modifiabilité* et la sécurité (savoir si les composants sont autorisés à accéder à certains types de données..).

Existe-t-il une définition permettant de nous stipuler ce qui est architectural ? Afin d'identifier cette définition, nous pouvons commencer par définir ce qu'est l'architecture.

Notre critère pour affirmer qu'un élément est architectural est qu'il doit être soit : un composant, une relation entre plusieurs composants, ou une propriété visible (des composants ou des relations) afin de pouvoir se pencher sur la capacité du système à répondre favorablement à ses exigences qualité ou à supporter la décomposition du système en plusieurs pièces indépendantes que l'on peut implémenter.

Voici quelques aspects essentiels de cette définition que nous avons identifiés [PW92, Bai12]:

- L'architecture décrit ce qui est dans le système. Lorsque nous déterminons notre contexte, nous déterminons une frontière qui décrit ce qui est dans le système et hors du système.
- L'architecture décrit la partie qui est dans le système et l'information qui est dans l'architecture est la description la plus abstraite mais la plus significative de cet aspect du système.
- L'architecture établit le lien entre les exigences liées à la spécification du système et le reste de la conception. Si nous (architectes) estimons que certaines informations sont essentielles pour améliorer la capacité du système à répondre à certaines exigences, alors nous pouvons considérer ces informations comme architecturales. À contrario, nous pouvons éliminer certains détails à condition que nous restions tout de même en mesure de composer une architecture qui pourra répondre aux exigences clés.
- Il est important de réaliser qu'une architecture se doit avant tout d'être compréhensible et que trop de détails dans l'architecture n'est jamais apprécié.
- Une architecture est contraignante dans la mesure où elle impose des conditions à toutes les spécifications de la conception de bas niveaux.

En somme, être de nature architecturale consiste à être la description la plus abstraite d'un système axé sur les exigences majeures de la spécification [Pri12].

Il n'est pas évident de trouver tous les aspects du système qui relèvent du domaine architectural et il est peu probable que nous arrivions à tous les lister. La spécification architecturale ne cesse d'évoluer car la technologie ne cesse d'évoluer. Nous devrons toujours continuer à déterminer ce qui est architectural et ce qui ne l'est pas afin que l'on puisse identifier les éléments les plus importants de notre système [Hsu01].

Corrélativement, l'architecture étant la base du logiciel et la spécification architecturale évoluant en permanence, c'est le système logiciel qui s'en retrouve fortement impacté ; ce dernier devient ainsi rapidement désuet car l'environnement dans lequel il a été développé, (de même qu'il en est pour la spécification architecturale) est devenu obsolète [Som11]. Dans son article [LHB85], Lehman affirme qu'un logiciel qui n'évolue pas devient progressivement inutile.

Le logiciel en question au sein de l'organisation est souvent le logiciel métier et le remplacer requiert des opérations liées au business très risquées. En effet, de nombreux anciens systèmes sont essentiels à l'organisation qui les utilise pour toutes leurs opérations. Ceux-ci incluent les informations et procédures business qui ont émergé pendant la vie du système. Le risque de renoncer à ces systèmes et de les récrire est très élevé.

Les anciens systèmes sont considérés à risque pour le business pour deux raisons :

- Ils comportent toutes les informations liées au business qui sont très difficiles à manier. Ils ne peuvent en aucun cas risquer de perdre toutes leurs données, c'est pourquoi énormément d'anciens systèmes sont toujours utilisés de nos jours.

- Maintenir ces systèmes est une entreprise très lourde et coûteuse et cela constitue un risque majeur dans la mesure où nous pourrions devenir inaptes à nous adapter aux exigences business modernes.

L'opération de moderniser une ancienne spécification architecturale ou un ancien système logiciel doit être établie à partir d'un constat réaliste, à travers duquel doit être déterminée la stratégie la plus appropriée pour évaluer ces anciens systèmes et leurs architectures logicielles afin de les comparer avec celui que l'on veut mettre en place. C'est ainsi que nous allons discuter dans la prochaine section de la méthode SACAM[12] qui permet de comparer des architectures logicielles afin de savoir quelle est celle pour laquelle nous devrions opter.

1.2.1 La méthode SACAM

Comparer les architectures logicielles pour les systèmes à grande échelle est une tâche difficile qui dépend de différents facteurs et de nombreux environnements. L'architecture logicielle est, comme nous l'avons vu précédemment, structurée selon des spécifications, contraintes et règles particulières mais sont souvent mal documentées. Ainsi, du fait du déficit d'informations les concernant, et puisqu'il faut changer d'architecture logicielle lorsque la notre est désuète, il existe un besoin majeur de développer une méthode permettant l'analyse de la comparaison d'architectures logicielles pour les systèmes critiques [CGB+02, BCK03].

Puisqu'un tel système repose essentiellement sur un ensemble de critères incluant ceux qui dérivent des objectifs d'une organisation, SACAM compare les architectures de systèmes logiciels (pas le code source) selon leurs métriques, mais en revanche, il ne résout pas les problèmes liés à l'architecture, tel que la maintenance et l'évolution du logiciel qui a été implémenté [Bac03].

Comme le montre la figure 1-2, les intervenants, lors de la modernisation des anciens systèmes, ont typiquement les options stratégiques suivantes :

---

[12] *Software Architecture Comparison Analysis Methods*

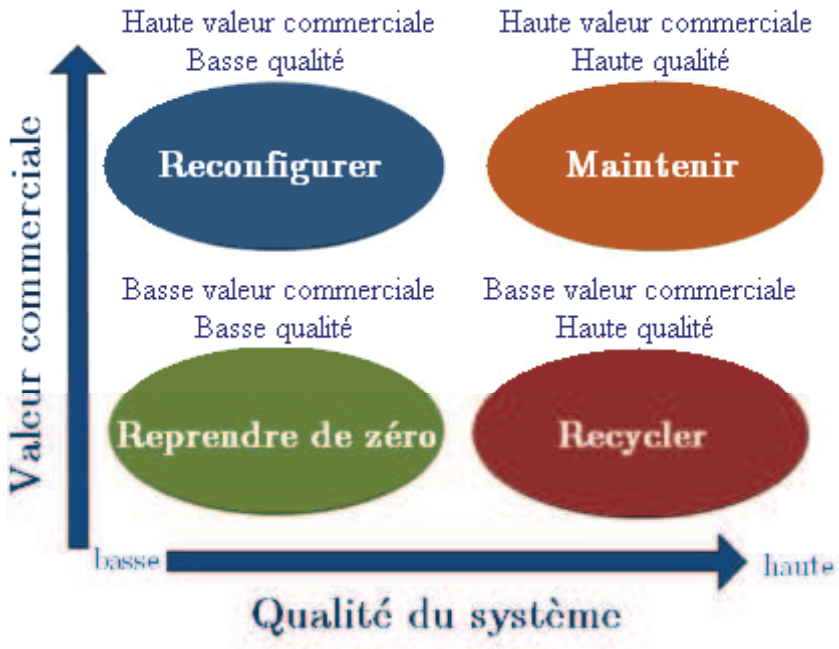

Fig. 1-2 - Quatre stratégies pour moderniser un système existant

- Retirer complètement le système et modifier les processus business afin qu'il ne soit plus nécessaire (Reconfigurer).
- Continuer à maintenir le système (Maintenir).
- Transformer le système en le repensant afin de l'améliorer (Recycler).
- Remplacer le système avec un nouveau système (Reprendre de zéro).

SACAM est utilisé dans l'analyse et la comparaison des produits logiciels et est utilisé dans le cadre de l'évaluation de la modernisation logicielle. Cette méthode fournit aux organisations concernées les processus de sélection de l'architecture en comparant les squelettes des architectures logicielles candidates pour le système visé [FV03, CKK02].

1.2.1.1 La recherche autour de SACAM

Il n'y a que peu de travaux de recherche liés à SACAM. Stoermer, Bachmann et Verhoef [SBV03] identifient et caractérisent une méthode d'analyse et comparaison d'architectures logicielles ayant pour but de comparer les architectures logicielles et d'en extraire les avantages et les limites dans le processus de développement ou de création d'un nouveau système logiciel.

De plus, ils présentent également la version généralisée de SACAM, qui est construite à partir d'un processus de comparaisons d'architectures candidates et de sélection de l'une d'entre elles. Cependant, c'est une opération différente de celle qui consiste à comparer de vrais cas d'études d'architectures logicielles pour atteindre les objectifs commerciaux d'une organisation, qui sert de base aux critères de comparaison. En soi, SACAM n'est pas approprié à un environnement "système critique". Corrélativement, cette méthode fournit un système d'évaluation se composant de trois gammes de valeur : non satisfaisant, pleinement satisfaisant ou satisfaisant sous certaines conditions en général [BEL+03].

SACAM a été développé la première fois par le Software Engineering Institute[13] (Institut de l'ingénierie logicielle) pour faire des comparaisons entre les architectures logicielles selon des multicritères extraits à partir des objectifs commerciaux d'une organisation.
SACAM fournit aux organisations concernées les processus de sélection de l'architecture en comparant les squelettes des architectures logicielles candidates pour le système visé [RG08].

Processus de comparaison : La comparaison est effectuée en utilisant une série d'étapes décrites ci-après :

- Étape 1 (Préparation). Pendant cette étape, les entrées nécessaires pour préparer une application réussie de la méthode sont disponibles. Celles-ci

[13] www.sei.cmu.edu

incluent (1) les architectures candidates (celles qui doivent être comparées) et (2) les objectifs commerciaux (la source des critères de comparaison).

- Étape 2 (Récupération des critères). Ici, un ensemble de critères pour la comparaison d'architectures est identifié. Un critère formule une condition pour que l'architecture soutienne les objectifs commerciaux de l'organisation. Les critères sont traduits en scénarios d'attributs qualité.

- Étape 3 (Identification des directives d'extraction). Durant cette étape, les vues architecturales, les tactiques, les styles et les modèles architecturaux que les évaluateurs recherchent pendant les extractions suivantes sont déterminés.

- Etape 4 (Extraction des vues et des indicateurs). Ici, les vues architecturales pour chaque candidat sont extraites selon les directives d'extraction de l'étape 3 ; cette étape détecte également les indicateurs qui soutiennent les scénarios d'attributs qualité de l'étape 2. Les techniques de récupération d'architecture peuvent être nécessaires pour produire des vues appropriées.

- Étape 5 (Évaluation). Dans cette étape, chaque critère est évalué pour une architecture candidate. L'évaluation est basée sur les preuves fournies par l'étape 4 et les scénarios d'attributs qualité identifiés. L'évaluation dessine l'aspect de l'architecture qui est le mieux adaptée aux critères identifiés.

- Étape 6 (Résumé). Cette étape récapitule les résultats de l'analyse.

1.2.1.2 Les résultats de SACAM

SACAM est un cadre analytique qui permet la comparaison de plusieurs architectures sans critères spécifiques. SACAM peut être utilisé avec quelques adaptations ou améliorations dans l'évaluation hâtive de l'architecture logicielle (avant son implémentation) et l'évaluation tardive de l'architecture logicielle (après son implémentation). Des scores sont donnés à chaque architecture logicielle candidate (aux scénarios, tactiques et vues architecturales), à partir desquels des recommandations concernant le processus décisionnel sont établies.

1.3 Les paradigmes architecturaux

Après avoir introduit le concept d'architecture logicielle, nous pouvons désormais nous concentrer sur les différents styles architecturaux notables que la communauté de l'ingénierie logicielle a identifiés. Nous pouvons distinguer de nombreux paradigmes architecturaux pour les systèmes distribués ; ainsi, ces dernières décennies ont vu les paradigmes objet, composant, agent et service coexister tout en ayant un développement parallèle [Oa14, Leg09].

Parmi ceux-là, trois ont fortement contribué à l'évolution des problématiques, il s'agit chronologiquement des architectures orientées objet, composant puis service; c'est ainsi que nous avons naturellement choisi de nous pencher en détails sur ces paradigmes (voir figure 1-3).

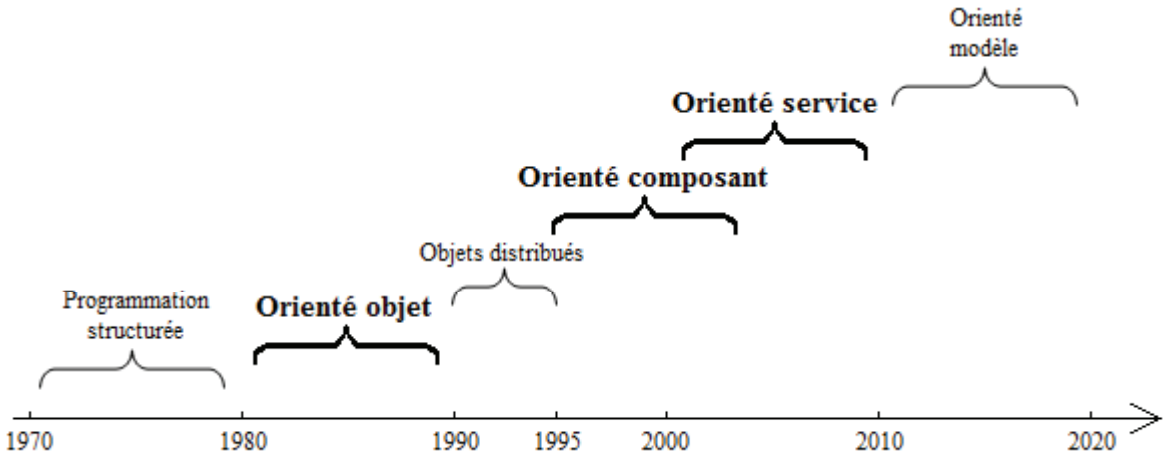

Fig. **1-3** - Évolution des paradigmes de développement (adaptée depuis [Oa14])

1.3.1 L'orienté objet

L'orienté objet (OO[14]) est un paradigme de conception et de programmation apparu au début des années 1960 avec le langage Simula (*Simulation of real systems*) qui a

[14] Orienté objet

été développé au *Norwegian Computing Center*. En 1970, Alan Kay et son groupe de recherche au *Xerox Park* ont développé le premier langage de programmation orienté objet : Smalltalk. Parallèlement aux travaux de Kay, Bjarne Stroustrup qui travaillait aux laboratoires Bell développait une extension du langage C, le C++ qui implémente les concepts de l'orienté objet. C'est en 1986 que prit davantage d'ampleur le paradigme orienté objet avec la programmation de la première conférence majeure lui étant exclusivement dédiée et l'apparition de douzaines de langages de programmations comme l'Objective-C, l'Eiffel, etc. [Oa14].

1.3.1.1 Paradigme objet

Le paradigme objet produit des systèmes qui sont des réseaux d'objets collaborant afin de répondre aux exigences du système [CL00].

Il consiste en la définition et l'interaction de briques logicielles appelées "objets" étant des unités conceptuelles [Oa99].

L'approche générique du paradigme orienté objet est de concevoir un système logiciel comme étant une collection d'entités interagissant appelées "objets". Chaque objet est défini par une identité, un état (décrit en termes de variables membres) et un comportement (décrit en termes de méthodes pouvant être invoquées).

L'unité basique de l'abstraction est l'objet, qui encapsule l'information relative à l'état (valeurs de variables d'instance) et le comportement (des méthodes qui sont fondamentalement des procédures).

Les attributs d'un objet lui permettent de maintenir un état, alors que ses méthodes lui permettent de fonctionner correctement [Oa14].

1.3.1.2 Les architectures logicielles à base d'objets

La modélisation par objet consiste à créer des diagrammes, des textes de spécification et du code basé sur les concepts objets pour décrire un système logiciel. Les langages de modélisation par objets sont des méthodes et des techniques pour

analyser et représenter graphiquement les systèmes logiciels. Il existe plusieurs méthodes de modélisation par objets (DOOS, MOT, OOSE ou encore OOD) ; cependant, de nos jours, la plupart de ces méthodes sont intégrées dans UML. L'architecture logicielle à base d'objets consiste à décrire un système comme étant une collection de classes (les entités à abstraire et l'encapsulation des fonctionnalités) qui peuvent avoir des objets (des instances) et communiquent entre eux par envois de messages.

1.3.2 L'orienté composant.

Le développement à base de composants est apparu dans les débuts des années 1960 et a été proposé par Mcllroy dès lors qu'il a implémenté une infrastructure sur Unix en utilisant des composants *pipelines* et des filtres [Oa05].

1.3.2.1 Paradigme composant

La motivation derrière l'utilisation des composants était initialement de réduire le coût de développement, mais il devint ensuite plus important de réduire le temps de mise sur marché et de rapidement satisfaire les requêtes du client. Actuellement, l'utilisation des composants est plus souvent motivée par des possibles réductions des coûts de développement. À l'aide des composants il est possible de produire plus de fonctionnalités avec le même investissement en termes de temps et d'argent.
Le développement à base de composants est également étroitement lié à la réutilisation, notamment pour palier aux défaillances ayant été observées avec le développement orienté objet. L'idée fondamentale est assez simple puisqu'elle consiste à utiliser des composants qui sont déjà développés lorsqu'il s'agit de développer de nouveaux systèmes. Beaucoup ont tenté de définir le terme de composant logiciel mais la définition qui a été adoptée par l'ensemble de l'ingénierie logicielle consiste à dire qu'un composant est une partie du logiciel sous une forme binaire (c.-à-d. qu'il n'est pas nécessaire de le reconstruire), avec des interfaces contractuellement spécifiques (c.-à-d. un API défini et toutes hypothèses dans lesquelles le composant peut fonctionner). Un composant peut être déployé

indépendamment (c.-à-d. il peut être chargé dynamiquement dans le système, ou être dynamiquement remplacé). Un composant doit avoir un mécanisme qui rend possible son intégration dans le système sans le modifier et le reconstruire.

1.3.2.2 Les architectures logicielles à base de composants

Les architectures logicielles à base de composants décrivent les systèmes comme un ensemble de composants (unités de calcul ou de stockage) qui communiquent entre eux par l'intermédiaire de connecteurs (unité d'interactions). Leurs objectifs consistent à :

- réduire les coûts de développement
- améliorer la réutilisation des modèles
- faire partager des concepts communs aux utilisateurs de systèmes
- construire des systèmes hétérogènes à base de composants réutilisables sur étagères.

Pour asseoir le développement de telles architectures, il est nécessaire de disposer de notations formelles et d'outils d'analyses de spécifications architecturales. Les langages de description d'architectures (ADL : *Architecture Description Language*) constituent une bonne réponse [Oa14].

1.3.3 L'orienté service.

Le paradigme orienté service est relativement récent ; il date du début des années 2000 et est dorénavant bien établi dans le domaine de l'ingénierie logicielle. Ce paradigme de développement logiciel est directement inspiré des modes d'organisations commerciales réelles entre multinationales, et se base sur leur notion classique de service offert.

1.3.3.1 Paradigme service

Le point central du paradigme orienté service est le concept de service, ainsi que la

stratégie à aborder pour pouvoir exposer les capacités du système aux consommateurs à travers une architecture orientée service. Dans cette optique, la technologie des Web services est la technologie la plus souvent répandue permettant d'implémenter efficacement le concept des services et des SOA. Les Web services utilisent essentiellement le XML[15] pour créer de robustes connexions.
Pour tenter de comprendre au mieux le paradigme orienté service, nous devons d'abord commencer par donner une définition claire du terme "service". Un service est une fonction qui est bien définie et qui forme un accord contractuel entre le fournisseur et le consommateur. Cette interaction entre le fournisseur et son client est faite par le biais d'un médiateur (qui peut être un bus) responsable de la mise en relation des participants. Les services sont généralement implémentés comme des entités logicielles à forte granularité et ne dépendent ni du contexte, ni de l'état d'autres services. Ils englobent et proposent les fonctionnalités des entités des systèmes. Ces systèmes peuvent aussi être définis comme des couches applicatives.

1.3.3.2 Les architectures logicielles orientées service

L'architecture logicielle orientée service est une discipline architecturale qui doit être utilisée de manière à construire des infrastructures permettant à ceux qui ont des besoins (les consommateurs) et ceux qui proposent leurs fonctionnalités (les fournisseurs) d'interagir via les services à travers des domaines disparates de la technologie. La première architecture orientée service fut mise en place avec l'utilisation des ORBs (*Object Request Brokers*) basée sur la spécification CORBA[16].
Les services agissent comme un noyau facilitant les inter-échanges de données électroniques entre ces deux acteurs. La communication entre ces services peut impliquer un simple retour de données ou une activité (coordination de plusieurs services) [Oa14].

La figure 1-4 montre une architecture orientée service de base. Nous y voyons un

[15] Extensible Markup Language
[16] http://www.corba.org/

consommateur de service sur la droite envoyant un message de demande de service à un fournisseur de services sur la gauche. Le fournisseur de services renvoie un message de réponse au consommateur de service. La demande de service et les connexions liées à la réponse sont définies de façon à être compréhensibles pour le consommateur et le fournisseur. Un fournisseur de services peut également être un consommateur de service.

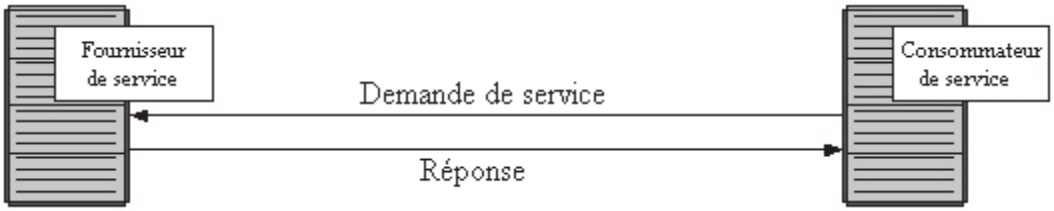

Fig. 1-4 - Architecture orientée service de base

L'architecture logicielle orientée service peut également disposer d'un annuaire de services que gère le service *broker* (voir figure 1-5) permettant de faire le lien entre le consommateur et le fournisseur de services qui s'ignorent. Les fournisseurs publient leurs services dans ces registres qui sont ensuite consultés par les consommateurs afin de déterminer ceux qui correspondent à leurs besoins [Kim08].

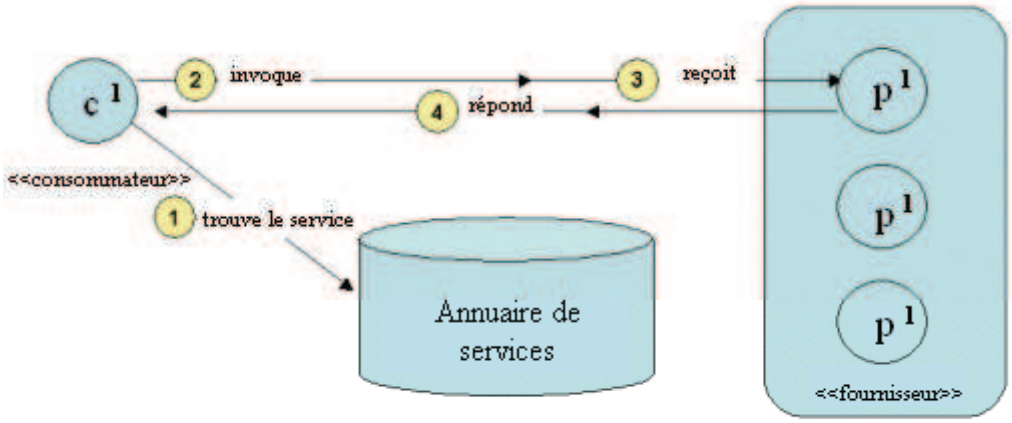

Fig. 1-5 - Architecture orientée service impliquant un annuaire de services

1.3.4 Pourquoi SOA ?

Nous nous sommes concentrés sur les SOA (ainsi que sur les technologies des services Web pour les implémenter) car ce sont les paradigmes architecturaux les plus largement utilisés ces dernières années. L'émergence de SOA qui fractionne les applications en fonctions et processus individuels, appelés services, présente aux organisations plusieurs nouveaux challenges intéressants.

L'orienté service est un paradigme de développement récent qui consiste à diviser les développeurs en deux entités collaboratives indépendantes : les développeurs de services (fournisseurs) et les services brokers (éditeurs). L'orienté service est une sorte d'évolution logique de l'orienté objet et de l'orienté composant. Le choix d'utiliser une approche SOA dans le développement de l'architecture intervient notamment afin de pallier aux défaillances majeures observées avec les deux paradigmes de développement précédemment cités.

Nous pouvons noter, pour l'orienté objet, d'importantes limites en termes de granularité, de niveau de réutilisation des objets (notamment du à leur fort couplage), la pauvre lisibilité de la structure des applications objets, la nature manuelle du mécanisme de gestion des objets et le faible nombre d'outils pour déployer les exécutables.

Les paradigmes de développement à base de composants fournissent quant à eux des modèles à un niveau élevé exclusivement, sans expliciter comment ces modèles peuvent être liés au code source et surtout, ces architectures restent un concept *ad hoc* connu presque qu'exclusivement par la communauté académique mais le monde industriel s'y intéresse de plus en plus. Toutes ces limites observées avec les technologies passées et l'aptitude qu'a l'orienté service à répondre favorablement aux exigences fonctionnelles et aux exigences liées aux attributs qualité (exigences non fonctionnelles) [CS09, Lap09] nous poussent à opter pour ce paradigme de développement logiciel [Kim08].

1.4 La qualité logicielle, concepts et mise en œuvre

La qualité d'un logiciel est l'un des enjeux majeurs dans la conception des outils logiciels [AL11a]. L'analyse et l'évaluation des logiciels sont deux phases ayant connu un essor très important dans la communauté des systèmes informatiques durant ces trois dernières décennies [PB93]. L'effort de développement, le temps et les coûts des systèmes complexes étant considérablement élevés, il est important d'assurer une qualité optimale au logiciel pour limiter les frais [AL11b].

Cette partie du manuscrit introduit une synthèse bibliographique liée à la qualité logicielle et a pour objet de faire un état de l'art et une liste non exhaustive des modèles de qualité les plus connus. Nous y abordons également la définition de la qualité logicielle vue par les spécialistes du domaine (qui pour certains aboutit à un modèle de qualité plus ou moins formel).

La partie liée à la qualité logicielle est structurée comme suit:

D'abord nous tenterons de définir le concept de qualité dans la section 1.4.1. Ensuite, nous donnerons quelques définitions initiales qui nous aideront à comprendre comment aborder ce mot évasif et subjectif, ainsi qu'une perspective élargie du concept de la qualité en présentant des aspects plus philosophiques sur ce qu'est la qualité. La section 1.4.2 traite également de la qualité à travers un aperçu des modèles de qualité et structures qualité les plus populaires de nos jours. Et finalement, nous conclurons cette section du manuscrit (section 1.5) avec une discussion sur les structures et modèles de qualité que l'on aura définis, avec un intérêt particulier pour le modèle de McCall et sa déclinaison : le modèle ISO/IEC 9126-1 qui ont servis de base pour l'ensemble de nos travaux.

1.4.1 Qu'est ce que la qualité ?

Pour comprendre l'univers de la qualité du logiciel, il est essentiel de répondre à la question très couramment posée : qu'est ce que la qualité ? Une fois que le concept

de la qualité est compris, il est plus facile de comprendre les différentes structures de la qualité disponibles sur le marché [HH01, KP96]. Avant d'étudier les différents modèles ou "structures de qualité" existants, il est essentiel qu'on puisse répondre à cette question fondamentale. Beaucoup d'auteurs et de chercheurs ont tenté d'apporter une réponse à cette question, et nous n'avons pas l'ambition de donner une nouvelle définition de ce qu'est la qualité ; nous répondrons plutôt à la question en étudiant les réponses que certains de ces auteurs et chercheurs de la communauté de l'ingénierie logicielle ont apportées.

L'étude des définitions des auteurs notoires est une perspective intéressante pour examiner le secteur des structures de qualité. Cette perspective fournit une alternative constructive sur la manière de concevoir les structures de qualité [HR96]. Ces définitions de gestion de la qualité peuvent parfois être une bonne alternative aux modèles plus formalisés de qualité que l'on présentera dans la dans la section 1-8.

1.4.1.1 La qualité selon Crosby

La notion de qualité selon Philip B. Crosby consiste à établir un système comme étant le plus conforme à la spécification possible [Cro79]. Cependant, il se concentre également sur la façon d'essayer de comprendre la longue liste d'attentes du client au sujet de la qualité en élargissant la perspective de production étroite de la qualité avec une perspective externe supplémentaire. Crosby souligne également qu'il est important de définir clairement la qualité pour pouvoir mesurer et contrôler le concept. Il résume sa perspective sur la qualité en quatorze étapes établies autour de quatre notions fondamentales de la gestion de la qualité :

1) La qualité est définie comme la conformité aux exigences de la spécification et non comme de "l'élégance".

2) Le système qualité des organisations qui tentent de répondre aux exigences des utilisateurs consiste à le faire correctement la première fois.

3) la norme performance doit être le "zéro défaut" et non pas le "qui s'en approche".

4) la mesure de la qualité est le coût de la qualité. Les coûts de l'imperfection, si elle est corrigée, exercent un bienfait immédiat, aussi bien sur la performance que sur les relations clients.

1.4.1.2 La qualité selon Deming

Un des points les plus forts de Deming consiste à dire que la qualité doit être définie en termes de satisfaction du client – qui est un concept bien plus large que celui de répondre à la conformité de la spécification.

La philosophie de Deming à propos de la qualité stipule que répondre aux exigences de l'utilisateur est la tâche que tous ceux dans l'organisation doivent s'atteler à accomplir.

En outre, le système de gestion doit permettre à chacun d'être responsable de la qualité de sa production. Pour mettre en application sa perspective sur la qualité, Deming a présenté ses quatorze "Points pour la Gestion" afin d'aider les organisations à comprendre et à réaliser la transformation nécessaire [Dem88]:

1. Améliorer constamment les produits et les services.
2. Adopter la nouvelle philosophie de la qualité totale.
3. Cesser les inspections interminables pour viser la qualité.
4. Réduire le coût total.
5. Améliorer constamment les processus de production et de service.
6. Instituer une formation pour le personnel de l'organisation.
7. Instituer une nouvelle forme de management pour faciliter le travail des hommes et des machines.
8. Chacun doit œuvrer pour la qualité du système.
9. Rompre les barrières entre les différents départements de l'organisation.

10. Éliminer les slogans et les objectifs qui poussent les employés à atteindre le "zéro défaut".
11. Éliminer les quotas de production, objectifs et travaux interprétés numériquement.
12. Ôter les obstacles qui empêchent le personnel d'être fier de son travail.
13. Instituer un vigoureux programme d'éducation et d'amélioration personnelle.
14. Concerner l'ensemble du personnel pour accomplir ces transformations.

1.4.1.3 La qualité selon Feigenbaum

La définition de Feigenbaum sur la qualité concerne indubitablement une réponse aux exigences du client [Fei83]. En fait, il va très loin dans sa définition de la qualité en soulignant l'importance de satisfaire le client dans ses attentes réelles et prévues. Feigenbaum précise essentiellement que la qualité doit être définie en termes de satisfaction du client, cette qualité est multidimensionnelle (elle doit être largement définie), et puisque les exigences ne cessent d'évoluer, la qualité doit également être un concept dynamique en perpétuelle évolution. Il est clair que la définition de Feigenbaum de la qualité n'englobe pas seulement la gestion du produit et des services mais aussi celle du client et de ses attentes.

1.4.1.4 La qualité selon Ishikawa

La perspective d'Ishikawa sur la qualité est également une définition répondant aux exigences du client car il associe assez fortement le niveau de la qualité aux exigences changeantes de chaque client [Ish85]. Il veut dire par là que la qualité est un concept dynamique puisque les besoins, les exigences et les attentes d'un client changent continuellement. La qualité doit être définie dynamiquement et intégralement. Ishikawa inclut également le concept de prix comme étant un attribut de la qualité - c.-à-d., qu'un produit vendu trop cher ne peut ni répondre à la satisfaction de l'utilisateur ni être de "bonne" qualité.

1.4.1.5 La qualité selon Juran

Juran prend une route quelque peu différente de celles des auteurs précédemment mentionnés lorsqu'il s'agit de définir la qualité. Son avis consiste à dire que nous ne pouvons pas employer le mot qualité lorsqu'il s'agit de satisfaire les attentes d'un utilisateur puisqu'elles sont très difficiles à atteindre. Juran définit plutôt la qualité comme étant "l'aisance à l'utilisation" [Jur88]. Il propose trois processus de gestion fondamentaux pour la tâche de la gestion de la qualité. Ces trois éléments sont :

Planification de la qualité : Un processus qui identifie les clients, leurs exigences, les caractéristiques du produit et du service que les clients attendent, et les processus qui fourniront ces produits et services avec les attributs corrects et ainsi facilitent le transfert de ces connaissances à la production de l'organisation.

Contrôle de la qualité : Un processus dans lequel le produit est examiné et évalué en fonction des exigences originelles du client. Les problèmes détectés sont alors corrigés.

Amélioration de la qualité : Un processus dans lequel les mécanismes nécessaires sont mis en place de façon à ce que la qualité puisse être atteinte continuellement.

1.4.1.6 La qualité selon Shewhart

La définition de la qualité du "*maitre*" Shewhart, (comme l'appelle Deming) date des années 1920 et bien qu'elle soit maintenant très ancienne, elle est toujours utilisée aujourd'hui lorsqu'une organisation vise une qualité optimale du système.

Sa définition consiste à souligner qu'il existe deux aspects communs à la qualité : Le premier implique la considération de la qualité d'un élément comme étant une réalité objective indépendante de l'existence d'un homme. L'autre aspect de la qualité concerne ce que l'on pense ou sent comme étant le résultat de la réalité objective ; en d'autres termes, il existerait un coté subjectif de la qualité. Sa définition est une réponse parfaite aux attentes du client et à la conformité de la spécification [She31].

1.4.1.7 Conclusion des définitions

A travers l'étude de ces différentes définitions, nous pouvons d'abord noter qu'il existe deux courants de pensées majeurs qui sont considérés et traduits par l'ensemble des auteurs lorsqu'il s'agit de définir la notion de qualité logicielle :

Conformité de la spécification : la qualité qui est définie comme étant une considération majeure des produits ou services dont les caractéristiques satisfont une spécification définie.

Réponse aux besoins des utilisateurs : la qualité qui est définie indépendamment des caractéristiques de la spécification. C'est-à-dire que la qualité est définie comme la capacité des produits ou service à répondre favorablement aux attentes des utilisateurs.

Tab. 1-1 - Croisement des courants de pensées et des définitions des auteurs

Courant / Auteur	Conformité de la spécification	Réponse aux besoins des utilisateurs
Crosby	X	
Deming		X
Feigenbaum		X
Ishikawa		X
Juran	X	
Shewhart	X	X

Tous les auteurs ont choisi d'axer leur définition autour d'un des deux courants, excepté Shewhart qui considère lui que ces deux courants sont essentiels et qu'il est important de les considérer dans ce qu'il estime étant sa définition de la qualité.

C'est sans doute pour cette raison que beaucoup d'experts considèrent cette définition de la qualité comme étant, encore de nos jours, la plus complète.

1.4.2 Modèles de qualité

Dans la section précédente nous avons présenté quelques auteurs notoires spécialistes de la gestion de la qualité ainsi que leurs perceptions de la qualité principalement parce que c'est une approche utilisée et appréciée pour traiter des questions de la qualité dans les organisations de développement logiciel. Considérant que les philosophies de gestion de la qualité présentées dans la section précédente représentent une vue de la qualité plus ou moins "flexible" dans la mesure où les recommandations établies dans chacune de ces définitions ne sont jamais applicables systématiquement de la même façon aux architectures et systèmes logiciels considérés, cette section présentera une vue de la structure qualité étant plus "concrète" à travers des modèles de qualité à appliquer toujours de la même manière pour viser une qualité optimale [BD02].

Il a été suggéré que la production logicielle soit hors de contrôle parce que nous ne pouvons pas quantitativement la mesurer. En effet, Tom DeMarco a déclaré que "*vous ne pouvez pas contrôler ce que vous ne pouvez pas mesurer*" [Dem86]. L'activité de mesure doit avoir des objectifs clairs et les secteurs doivent chacun être mesurés séparément pour assurer la bonne gestion du logiciel. Par exemple, nous savons que l'administrateur doit mesurer la qualité logicielle afin de comparer des projets, faire des prévisions et fixer des objectifs raisonnables d'amélioration. Afin de réaliser ces opérations, la communauté scientifique utilise des modèles de qualité présentant ce que nous appelons " la décomposition d'attributs qualité".

1.4.2.1 Modèle de McCall

L'un des modèles de qualité les plus connus, précurseur de ceux d'aujourd'hui, est le modèle de qualité présenté par Jim McCall et al. [MRW77, McC70] (également connu comme "General Electrics Model" de 1977). Ce modèle, ainsi que les autres modèles contemporains proviennent de l'armée américaine (ils ont été développés

pour la "US Air Force") et sont principalement destinés aux développeurs système et au processus de développement du système. Le modèle de qualité de McCall essaye d'établir le lien entre les utilisateurs et les développeurs en se concentrant sur un certain nombre d'attributs qualité du logiciel qui reflètent les vues des utilisateurs et les priorités des développeurs. Le modèle de qualité de McCall a, comme l'indique la figure 1-6, trois perspectives importantes pour définir et identifier la qualité d'un produit logiciel : révision du produit (capacité à subir des changements), transition du produit (adaptabilité à de nouveaux environnements) et opérations du produit (ses caractéristiques d'opération).

La révision du produit inclut la maintenabilité (l'effort requis pour détecter et réparer une anomalie du programme dans son environnement de fonctionnement), la flexibilité (la capacité à apporter des modifications requises par des changements du système d'exploitation) et la testabilité (la capacité à tester le programme, pour s'assurer qu'il ne contient pas d'erreurs et qu'il respecte la spécification). La transition du produit concerne la portabilité (l'effort requis pour transférer un programme d'un environnement à un autre), la réutilisabilité (la capacité à réutiliser le logiciel dans un contexte différent) et l'interopérabilité (l'effort requis pour coupler le système à un autre système). La qualité des opérations du produit dépend de l'exactitude (le degré auquel un programme respecte ses spécifications), la fiabilité (la capacité des systèmes à ne pas échouer), l'efficacité, l'intégrité (la protection du programme contre l'accès non autorisé) et la facilité d'utilisation (la facilité/ l'ergonomie du logiciel).

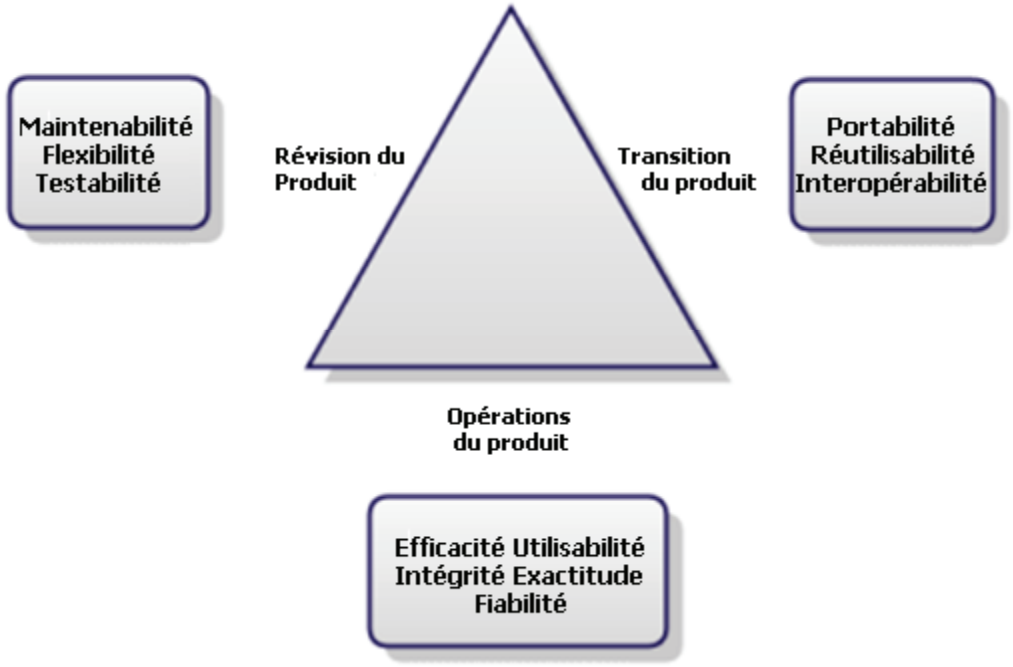

Fig. 1-6 - Le modèle de qualité de McCall (appelé aussi Triangle qualité de McCall) organisé autour de trois types de caractéristiques qualité

Le modèle détaille en outre les trois types de caractéristiques qualité (attributs qualité) dans une hiérarchie organisée autour de facteurs, critères et métriques :

- 11 Facteurs qualité (pour spécifier) : Ils décrivent la vue externe du logiciel, vue par les utilisateurs.
- 23 critères qualité (pour construire) : Ils décrivent la vue interne du logiciel, vue par le développeur.
- De nombreuses métriques qualité (pour contrôler) : Elles sont définies et utilisées pour fournir une échelle et une méthode de mesure.

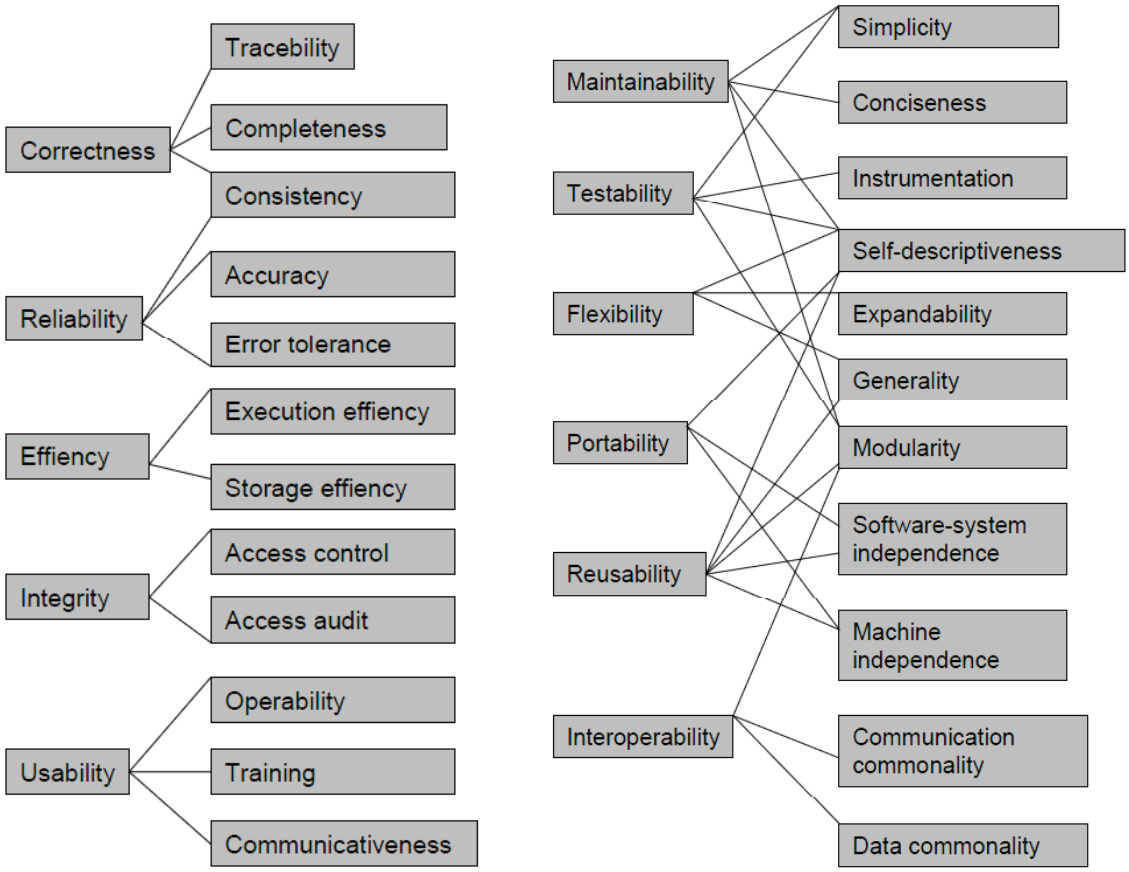

Fig. 1-7 - Le modèle de qualité de McCall hiérarchisé en facteurs et en critères.

Les facteurs qualité décrivent différents types de caractéristiques comportementales du système, et les critères qualité sont des attributs appartenant à un ou plusieurs facteurs qualité. La métrique qualité, a pour objectif de capturer certains des aspects d'un critère qualité. L'idée derrière le modèle de qualité de McCall est que les facteurs qualité puissent fournir une structure complète de la qualité du logiciel. A travers la figure 1-7, nous pouvons voir qu'il est possible que chacun des 23 critères qualité puisse composer plusieurs facteurs qualité (exemple de la "modularité" qui compose les facteurs qualité "maintenabilité, testabilité, réutilisabilité et interopérabilité") et que réciproquement, chacun des 11 facteurs qualité puisse être composé de plusieurs critères (le facteur qualité "efficacité" composé des critères qualité "efficacité d'exécution et efficacité du stockage").

1.4.2.2 Modèle de Boehm

Le deuxième précurseur des modèles de qualité d'aujourd'hui est le modèle de qualité présenté par Barry W. Boehm [BBK+78]. Boehm répond aux points faibles des modèles de qualité contemporains qui évaluent automatiquement et quantitativement la qualité du logiciel.

Ses tentatives de modèles consistent à définir qualitativement la qualité du logiciel par un ensemble donné d'attributs qualité. Le modèle de Boehm est semblable au modèle de qualité de McCall parce qu'il présente également un modèle de qualité hiérarchisé, structuré autour des caractéristiques de haut niveau, de caractéristiques de niveau intermédiaire, et de caractéristiques primitives - qui impactent la qualité globale (voir figure 1-8).

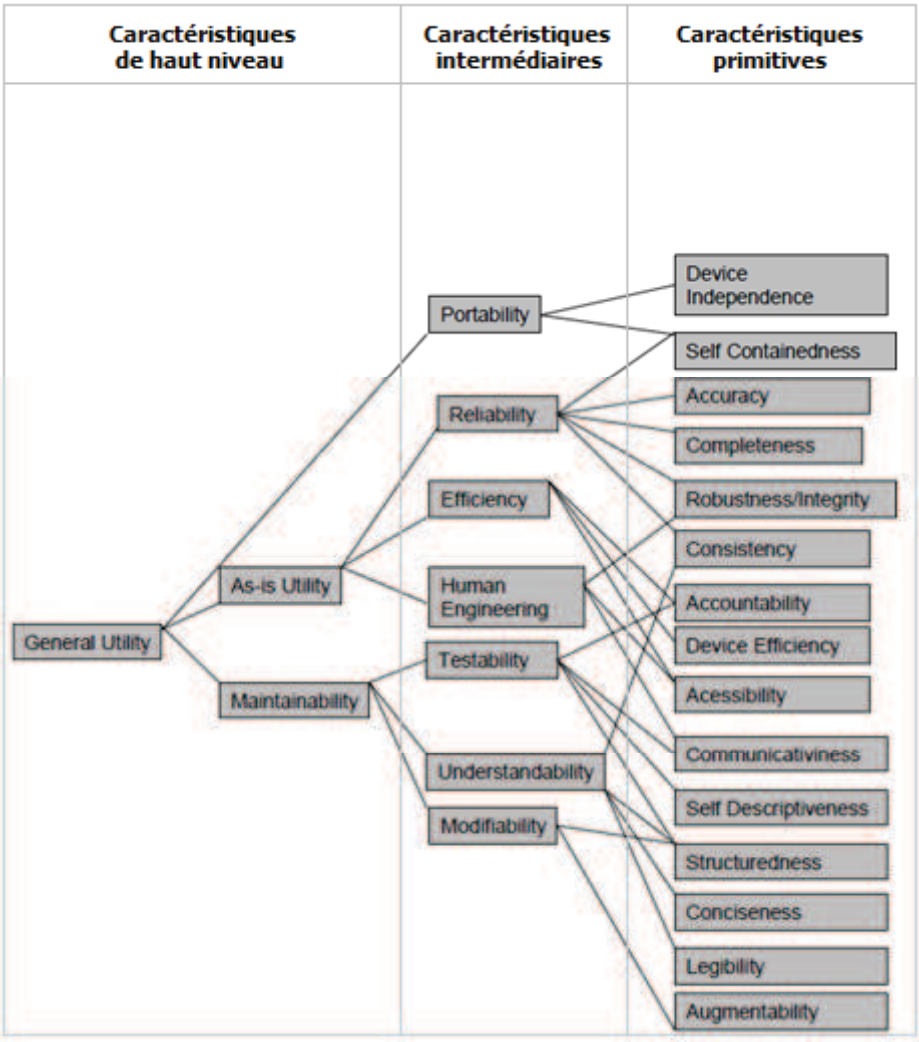

Fig. 1-8 - Arbre des caractéristiques qualité du logiciel de Boehm.

Les caractéristiques de haut niveau représentent les exigences basiques de haut niveau sur lesquelles l'évaluation de la qualité du logiciel pourrait être menée (l'utilité générale du logiciel). Les caractéristiques de haut niveau abordent trois questions principales qu'un utilisateur du logiciel peut se poser [BBK+78]:

- **Utilité réelle** : À quel point (facilement, sûrement, efficacement) puis-je utiliser le logiciel.
- **Entretien** : A quel point est-il facile de comprendre, modifier et re-tester ?
- **Maintenabilité** : Puis-je toujours l'utiliser si je change d'environnement ?

La caractéristique de niveau intermédiaire représente les 7 facteurs qualité de Boehm qui représentent ensemble les qualités attendues d'un système logiciel : [BBL76]:

- **Portabilité** (caractéristiques d'utilisation générale) : Le code possède la caractéristique "portabilité" dans la mesure où il peut être utilisé facilement et correctement sur d'autres configurations que la configuration actuelle.
- **Fiabilité** : Le code possède la caractéristique "fiabilité" dans la mesure où il compte remplir ses fonctions attendues d'une manière satisfaisante.
- **Efficacité** : Le code possède la caractéristique "efficacité" dans la mesure où il accomplit ses objectifs sans gaspillage de ressources.
- **Facilité d'utilisation** : Le code possède la caractéristique "facilité d'utilisation" dans la mesure où il est fiable et efficace.
- **Testabilité** : Le code possède la caractéristique "aptitude à l'essai" dans la mesure où il facilite l'établissement des critères de vérification et supporte l'évaluation de sa performance.
- **Compréhensibilité**: Le code possède la caractéristique "compréhensibilité" dans la mesure où son but est assez clair pour l'inspecteur.
- **Flexibilité** (modificabilité): Le code possède la caractéristique "modificabilité" dans la mesure où il facilite l'incorporation de changements, une fois que la nature du changement désiré a été déterminée.

La structure de plus bas niveau dans la hiérarchie de caractéristiques dans le modèle de Boehm est la hiérarchie des caractéristiques primitives des métriques. Les caractéristiques primitives fournissent la base pour définir les métriques qualité (qui représentait l'un des buts de Boehm quand il a construit son modèle de qualité). Par conséquent, le modèle présente une ou plusieurs métriques censées mesurer une caractéristique primitive donnée voir figure 1-8.

Bien que les modèles de Boehm et de McCall aient l'air d'être très ressemblants, la différence réside dans le fait que le modèle de McCall se concentre principalement sur la mesure précise des caractéristiques de haut niveau "utilité réelle", tandis que le

modèle de qualité de Boehm est basé sur un éventail de caractéristiques avec avant tout un intérêt étendu et détaillé sur la "maintenabilité".

1.4.2.3 Modèle de FURPS/FURPS+

Le modèle FURPS, originellement présenté par Robert Grady [Gra92] (et étendu par Rational Software, et maintenant IBM Rational Software en FURPS+3 [Rat03]) est structuré de la même façon que les modèles de qualité précédents mais est tout de même de renommée moindre. FURPS considère les attributs qualité suivants :

- **Fonctionnalité** - inclut des ensembles de caractéristiques, les capacités et la sécurité.
- **Facilité d'utilisation** - inclut des facteurs humains, l'esthétique, la cohérence dans l'interface utilisateur, l'aide en ligne, la documentation pour l'utilisateur, et les équipements pour la formation.
- **Fiabilité** - inclut la fréquence et la sévérité de l'échec, la "récupérabilité", la prévisibilité, l'exactitude, et le temps moyen entre l'échec (les moyennes des temps de bon fonctionnement).
- **Performance** - impose des conditions liées aux exigences fonctionnelles telles que la vitesse, l'efficacité, la disponibilité, l'exactitude, le débit, le temps de réponse et le temps de rétablissement.
- **Pouvoir de support** - inclut la testabilité, l'extensibilité, l'adaptabilité, le pouvoir d'entretien, la compatibilité, le pouvoir de configuration, la capacité du service.

Les catégories FURPS sont de deux types différents : fonctionnel (F) et non fonctionnel (URPS). Ces catégories peuvent être employées comme exigences du produit ou comme évaluation de la qualité du produit.

1.4.2.4 Modèle de Dromey

Un modèle semblable aux modèles de McCall, Boehm et FURPS (+) mais bien plus récent, est le modèle de qualité présenté par R. Geoff Dromey [Dro95]. Dromey propose un modèle de qualité basé sur le produit qui reconnait que l'évaluation de la qualité diffère pour chaque produit et qu'une idée plus dynamique pour modéliser le processus est nécessaire. Dromey se concentre aussi bien sur les relations entre les attributs qualité et les sous attributs, que sur le moyen de connecter les propriétés logicielles du produit aux attributs qualité du logiciel.

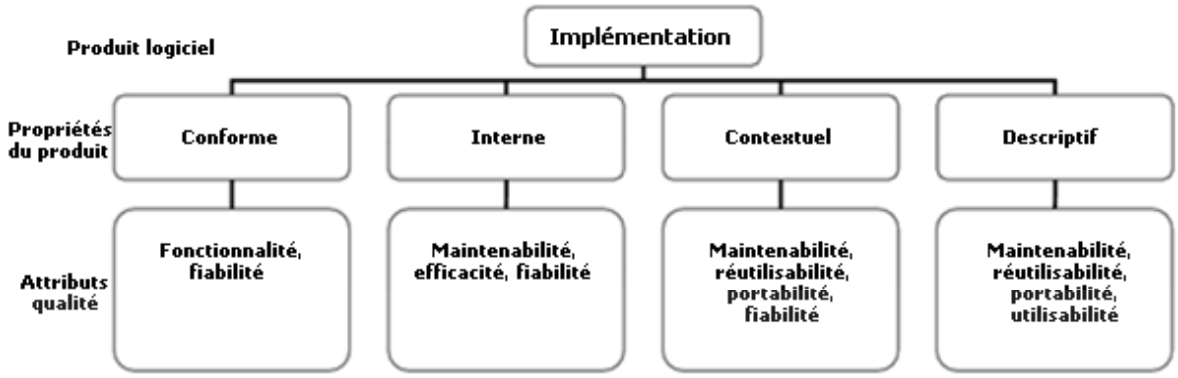

Fig. 1-9 - Principes du modèle de qualité de Dromey (adaptée à partir de [Dro96])

Comme l'illustre la figure 1-9, il existe trois éléments principaux dans le modèle générique de qualité de Dromey.

1. Les propriétés du produit qui influencent la qualité.
2. Les attributs qualité de haut niveau.
3. Les moyens de lier les propriétés du produit avec les attributs qualité.

Le modèle de qualité de Dromey est structuré autour d'un processus de cinq étapes :

1. Choisir un ensemble d'attributs qualité de haut niveau nécessaires pour l'évaluation.
2. Lister les composants et module du système.

3. Identifier les propriétés qualité pour les composants et modules (qualité du composant qui ont le plus d'impact sur les propriétés du produit).
4. Déterminer comment chaque propriété agit sur les attributs qualité.
5. Evaluer le modèle et identifier les défaillances.

1.4.2.5 Modèle ISO/IEC 9126-1

L'acronyme ISO (pour International Organization for Standardization) de renommée mondiale désigne comme son nom l'indique une organisation responsable d'un éventail de normes et standards dont l'ISO/IEC 9126-1 consacrée à l'évaluation du logiciel (Caractéristiques et directives qualité pour une utilisation standard) [Iso01].

Cette norme a été basée sur le modèle de McCall. Elle est structurée fondamentalement de la même façon que ce modèle [CF06].

Nicole Lévy stipule dans [Lév14], que L'ISO/IEC 9126-1 et sa mise à jour ISO/IEC 25010, qui fait partie de la série des normes SQuaRE [SAA03], définissent des propriétés de qualité qui peuvent être utilisées pour décrire un produit logiciel.

Comme nous le montre la figure 1-10; L'ISO/IEC 9126-1 propose un standard qui répertorie six domaines d'importance majeure que l'on appelle les facteurs qualité pour l'évaluation logicielle : la fonctionnalité, la portabilité, la fiabilité, la maintenabilité, l'efficacité et *l'"utilisabilité"* sont ces facteurs qualité, et cette norme identifie aussi des caractéristiques-qualité internes et externes du logiciel.

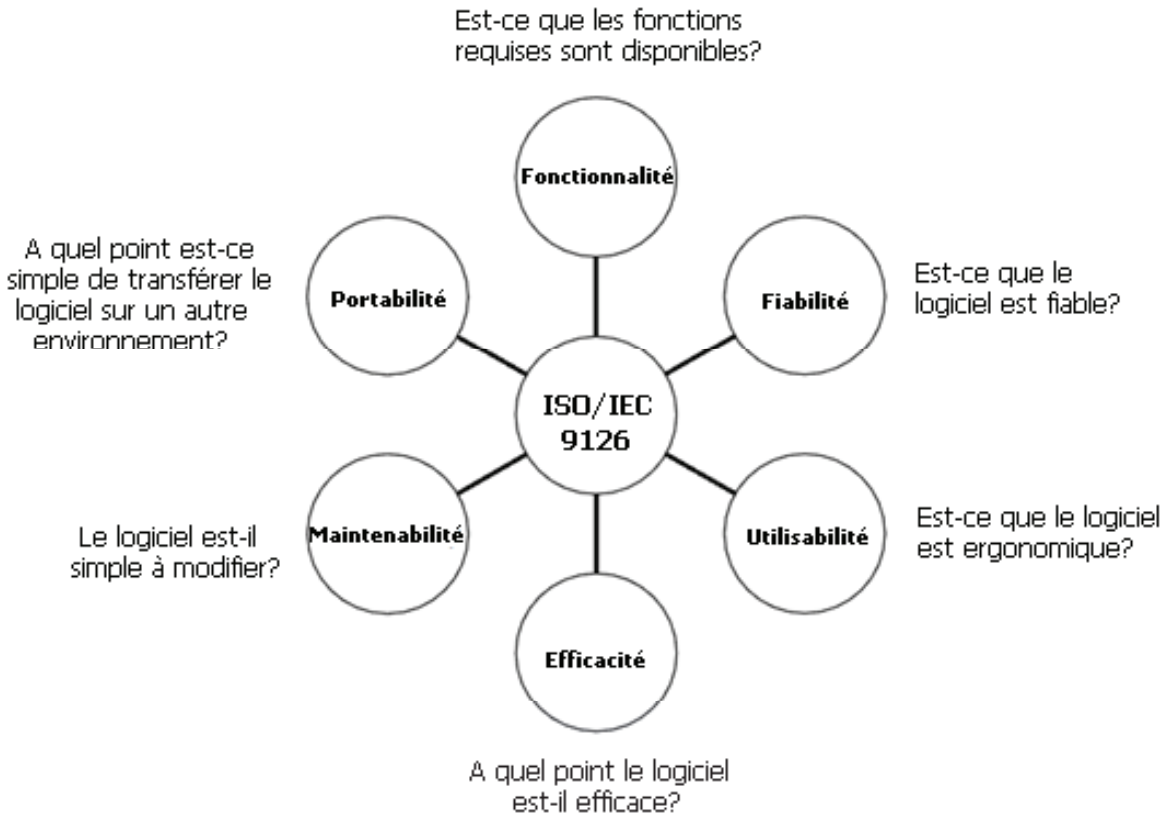

Fig. 1-10 - Le modèle qualité ISO/IEC 9126-1

Chacun de ces six facteurs qualité, définissent des sous-facteurs, ou critères qualité que nous pouvons distinguer dans la figure 1-11 suivante :

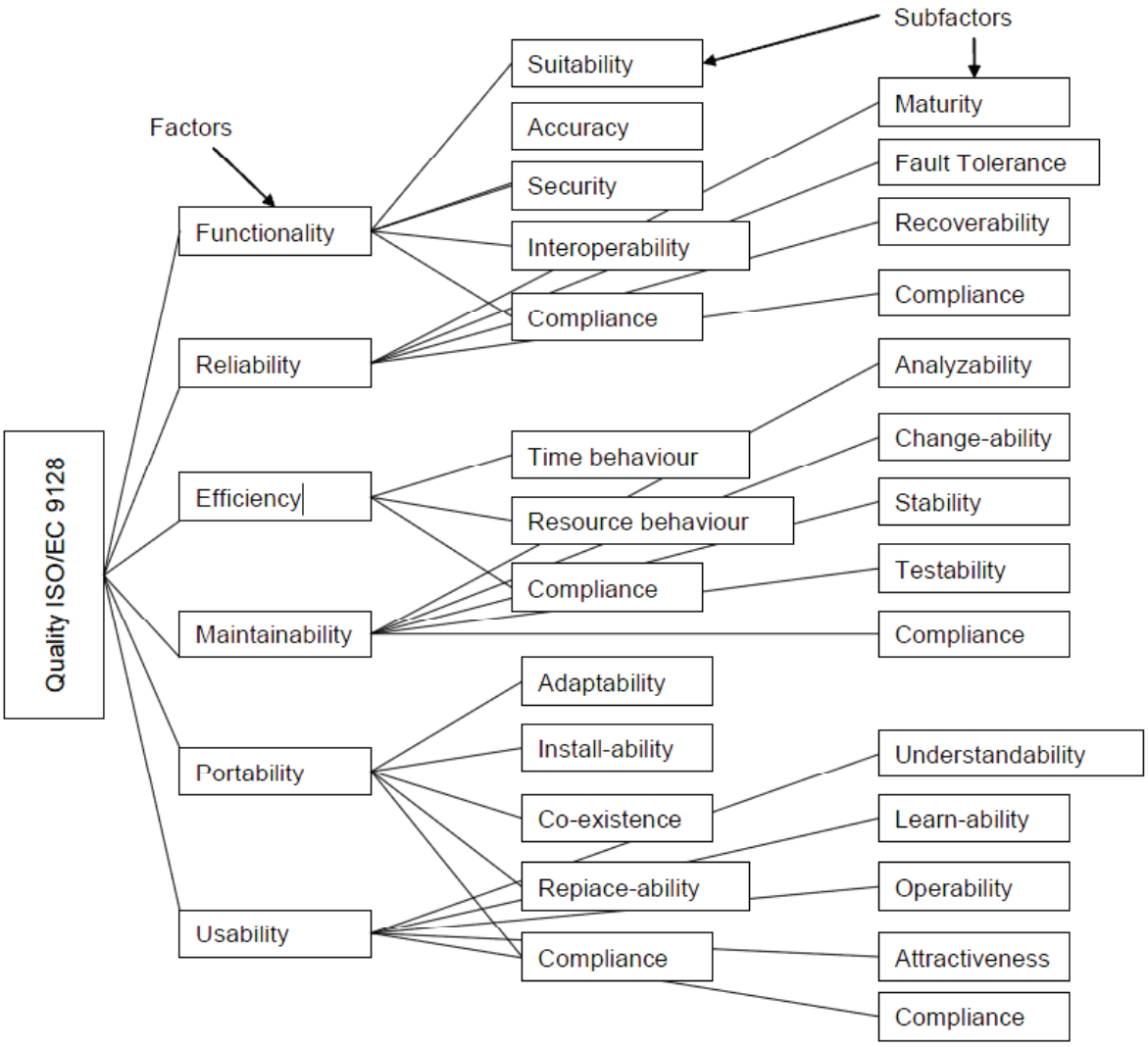

Fig. 1-11 - ISO/IEC 9126-1 : Évaluation du logiciel : Attributs qualité

Nota Bene : Malgré que nous nous soyons fortement inspirés du modèle de la norme ISO/IEC 9126-1 pour établir nos travaux, nous ne définissons pas chacun des attributs du modèle de la figure 1-11 car nous travaillerons pour notre solution avec des attributs différents qui sont exclusivement d'ordre technique et que nous présenterons en détails à partir de la section 3.2.2. L'intérêt d'exposer les informations relatives à ce modèle consiste à montrer que nous nous sommes réellement basés sur ce modèle de qualité ISO/IEC 9126-1 (étant lui-même une

déclinaison du modèle de McCall), notamment au niveau de la manière de hiérarchiser les attributs qualité relatifs à nos travaux.

1.5 Conclusion

A travers tout ce chapitre l'ambition a été d'examiner brièvement différentes structures de qualité avec une focalisation accentuée sur le modèle de McCall et le modèle ISO/IEC 9126-1[CF06] (ou encore sa mise à jour SQuaRE ISO/IEC 25010:2011 [SAA03]). L'idée était de nuancer et de fournir une vue simplifiée de ce qu'on labellise "qualité". Ce chapitre a prouvé que la qualité peut être un concept très évasif qui peut être approché d'un certain nombre de perspectives. Garvin [Gar84] a tenté de trier les différentes vues sur la qualité. Il a établi l'organisation des vues suivantes:

- Vue transcendantale, où la qualité est identifiée mais pas définie. La vue transcendantale est subjective et non quantifiable. Elle a souvent comme conséquence un logiciel qui dépasse les attentes de client.
- La vue utilisateur sur la qualité prend son point de départ dans un logiciel qui répond aux besoins de l'utilisateur. La fiabilité (taux d'échec, moyenne des temps de bon fonctionnement), la Performance/efficacité (temps pour effectuer une tâche), la maintenabilité et la facilité d'utilisation sont des problématiques dans cette vue.
- La vue de fabrication sur la qualité se concentre sur la conformité à la spécification et la capacité des organisations à produire le logiciel selon le processus logiciel. Ici la qualité du produit est atteinte à travers la qualité du processus. La réduction des pertes, le zéro défaut, "le bon logiciel la première fois" sont les concepts que cette vue abrite habituellement.
- La vue du produit sur la qualité spécifie habituellement que les caractéristiques du produit sont définies par les caractéristiques de ses sous

parties, par exemple taille, complexité, et couverture des tests. Les mesures de complexité du module, les mesures de conception et de code…etc.

- La vue basée sur la valeur de la qualité mesure et produit du rendement en équilibrant les exigences, le budget et le temps, le coût et le prix, et en fournissent des dates (délai d'exécution, temps calendaire), la productivité etc.

La plupart des modèles de qualité présentés dans ce chapitre pourraient convenir à la vue utilisateur, la vue de fabrication ou la vue du produit. Les modèles présentés ci-dessus sont soit focalisés autour des processus soit autour du niveau d'aptitude (ISO) où la qualité est mesurée soit en termes d'adhérence au processus, soit en termes de niveau d'aptitude, ou alors en un ensemble d'attributs utilisés pour évaluer la qualité (McCall, Boehm…) en faisant de la qualité un concept quantifiable.

Bien qu'ils présentent quelques avantages (en termes de mesurabilité objective), les modèles de qualité ramènent la notion de qualité à une série d'attributs statiques.

Cette structure de qualité est en grande contradiction avec les perspectives dynamiques d'exigences de l'utilisateur qui sont en perpétuelle évolution que certains auteurs notoires ont présenté. Il est facile de constater que les modèles de qualité représentent des perspectives plus étroites sur la qualité que les philosophies de gestion présentées par les auteurs et chercheurs.

L'avantage des modèles de qualité est qu'ils sont plus simples à mettre en place et à utiliser tandis que l'avantage des philosophies de gestion de la qualité est qu'ils ont probablement plus saisi le concept de la qualité.

Le modèle de McCall, qui décrit la qualité du logiciel et qui, comme nous l'avons montré, a mené à la norme internationale pour l'évaluation de la qualité du logiciel, ISO/IEC 9126-1 sert de base à notre travail. Notre modèle SOAQE est organisé autour des mêmes familles d'attributs qualité que pour ces modèles, à savoir facteurs qualité, critère qualité et métrique qualité auxquels on rajoute un plus haut niveau d'abstraction, les points de vue qualité. Chaque point de vue est divisé en plusieurs facteurs, critères et métriques. La méthode consiste à décomposer l'architecture

entière selon le modèle de McCall, c.-à-d. en une série d'attributs qualité afin de l'évaluer. Le résultat obtenu est une liste de points de vue qualité (dressés après avoir étudié les besoins commerciaux de l'organisation), eux même décomposés en facteurs regroupant des critères composés par des métriques. Nous examinerons notre modèle en détail dans la section 3.2.

Chapitre 2

L'ÉVALUATION DES ARCHITECTURES LOGICIELLES, UNE PROBLÉMATIQUE DÉLICATE

2.1 Introduction

La mesure précise est un préalable à toutes les disciplines de l'ingénierie, et la technologie de la programmation n'est pas une exception. L'objectif majeur recherché consiste à trouver des méthodes, des outils, permettant de développer des produits logiciels de haute qualité à un prix raisonnable.

Corrélativement, puisque les ordinateurs sont de nos jours utilisés dans presque tous les départements majeurs d'une organisation, la qualité des logiciels devient alors un facteur clé dans le succès de l'organisation.

Pendant des décennies, les ingénieurs et les chercheurs ont tenté d'exprimer les caractéristiques du logiciel avec des nombres afin de faciliter l'évaluation de la qualité du logiciel.

Il peut arriver qu'un logiciel soit délivré en temps et en heure, qu'il remplisse correctement et efficacement toutes ses fonctions et que ses coûts soient ce qu'on avait budgétisé mais qu'il ne nous satisfasse tout de même pas pour plusieurs raisons [HH01, KP96, Hab94] :

1. Il peut être difficile de comprendre le logiciel et difficile de le modifier. Ceci mène très souvent à des coûts répétitifs pour gérer la maintenance du logiciel, et ces coûts ne sont que très rarement insignifiants. Par exemple, l'article d'Elshoff [Els84] indique que 75% d'effort du logiciel de « General Motors » est consacré à la maintenance du logiciel et que c'est un chiffre que l'on retrouve assez souvent dans les industries de cette taille.

2. Il peut être difficile d'utiliser le logiciel ou à contrario très facile d'en faire mauvaise usage. Un rapport du GAO[17] [Gov77] (bureau de responsabilité du gouvernement américain) a identifié des frais entrepris par le gouvernement américain qualifiés d'inutiles et estimés à plus de dix millions de dollars pour

[17] Government Accounting Office

pallier à des problèmes d'ADP[18] suite à une mauvaise utilisation de leurs solutions logicielles.

3. Le logiciel peut être difficile à intégrer aux autres programmes du système. Ce problème qui est très contraignant ne cessera de prendre de l'ampleur puisque les types de machines continuent sans cesse de proliférer.

2.2 Pourquoi évalue-t-on la qualité logicielle ?

Une architecture inappropriée précipitera un projet à sa perte. Les objectifs liés à la performance ne seront pas atteints et ceux liés à la sécurité ne seront même pas considérés. L'utilisateur sera mécontent car la fonctionnalité désirée n'est pas disponible et qu'il est trop difficile de modifier le système pour l'ajouter. Les plannings et budgets ne seront également pas respectés et des mois ou des années plus tard, les changements qui auront été anticipés seront finalement abandonnés car ils deviendront en réalité trop coûteux [BCK03].

L'architecture détermine également la structure du projet : les bibliothèques de contrôle de configuration, les plannings et budgets, les objectifs de performance, la structure de l'équipe, l'organisation des documentations, et les activités de tests ou de maintenance sont toutes organisées autour de l'architecture. S'il s'avère qu'elle doive être modifiée en raison d'une éventuelle défaillance découverte tardivement, le projet entier peut être remis en cause. Il est nettement préférable de modifier l'architecture avant que quoi que ce soit n'ait encore été commencé afin de limiter les coûts d'une éventuelle reconstruction du projet [Zha99].

Jusque récemment, il n'existait pas de méthodes d'utilité générale pour valider une architecture logicielle. Les approches de validation étaient ad hoc et ne pouvaient pas être répétées. Pour ces raisons, elles n'étaient pas dignes de confiance et donc

[18] Adaptive Dynamic Programming

rapidement laissées à l'abandon. Mais depuis ce temps, les choses ont nettement évolué [Kaz01].

L'évaluation d'architecture est une manière bon marché d'éviter la catastrophe et elle n'ajoute que quelques jours de plus au planning initial. Elle peut être réalisée durant deux phases du cycle de vie d'un logiciel.

L'architecture logicielle peut être évaluée avant son implémentation (évaluation hâtive) ou après son implémentation (évaluation tardive).

L'évaluation précoce de l'architecture logicielle peut être effectuée à partir des spécifications, de la description de l'architecture logicielle et d'autres sources d'information, telles que les entretiens avec des architectes. L'évaluation tardive de l'architecture logicielle est effectuée à partir de métriques et peut aussi bien être utilisée pour évaluer les systèmes existants avant de futures améliorations ou opérations liées à la maintenance du système que pour identifier les points forts et faibles de l'architecture.

Les outils et les approches existantes ont montré leurs limites pour SOA. Ces dernières années, nous avons assisté au développement de nouvelles méthodes non automatisées développés par les industriels (ATAM, SAAM, etc.) [CKK01].

Ces méthodes, que nous allons succinctement présenter dans ce manuscrit de thèse doivent de préférence être appliquées à une architecture étant en phase de spécification, donc relativement tôt dans la vie du logiciel, mais naturellement, elles peuvent également être appliquées plus tard. Elles requièrent toutes une assemblée d'intervenants techniques pour les sessions de brainstormings, de présentations ou d'analyses que nous présentons ci-après.

2.2.1 Intervenants

Afin d'évaluer une architecture, les intervenants doivent exprimer leurs inquiétudes et montrer comment celles-ci peuvent être effacées. Un large panel d'intervenants fait chuter le risque de "survoler" les importantes problématiques architecturales. Le choix des intervenants pour une évaluation dépendra essentiellement des besoins de

l'organisation. Les catégories d'intervenants techniques sont souvent les mêmes pour les systèmes SOA, notamment lorsqu'ils travaillent avec des attributs de la même famille.
Voici au minimum la liste des intervenants techniques qui devront participer à l'évaluation d'une architecture [KKC00]:

2.2.1.1 Producteurs du système

1. Architectes logiciels

   Leurs responsabilités premières consistent à expérimenter et décider entre les différentes approches architecturales. Ils créent les interfaces et les spécifications. Ils valident l'architecture en fonction des exigences fonctionnelles et non fonctionnelles. Les architectes créent la documentation qui articule la vision architecturale aux autres évaluateurs et identifient les risques et les compromis de l'architecture ainsi que les raisonnements et les décisions de conception. Enfin, les architectes logiciels s'assurent que l'implémentation soit conforme à l'architecture.

2. Développeurs.

   Leur rôle principal concerne l'implémentation des éléments architecturaux du système en fonction de la spécification de l'architecture. Les développeurs offrent leur expertise pendant les différents processus de conception et ils dirigent les phases de test tout en créant les prototypes pour valider une approche architecturale.

3. Régulateurs d'utilisation de service.

   Leurs fonctions incluent la création de politiques pour l'utilisation des services, tels que spécifier que les services doivent être conformes à certains standards, et éventuellement définir des contraintes sur les services qui peuvent être utilisés. Ils doivent également définir des accords au niveau des services entre les organisations.

4. Testeurs.

Leurs responsabilités concernent la planification des tests des systèmes, l'exécution de l'ensemble des tests planifiés, l'enregistrement des résultats de tests et le report des défaillances observées.

5. Intégrateurs.

   Les intégrateurs s'assurent que l'architecture et l'implémentation soient conformes aux standards universels. Ils mettent en avant les approches architecturales qui simplifient l'intégration de service, les mises à jour et les remplacements

6. Développeurs chargés de maintenance.

   Leurs fonctions incluent la modification du logiciel pour corriger les défaillances et adapter le logiciel lorsque des changements environnementaux apparaissent (hardware ou système d'exploitation par exemple).

7. Chefs de projet.

   Leurs principales missions concernent essentiellement la gestion des efforts de développement, la création d'un plan de projet et le suivi de l'évolution du projet.

8. Directeur des systèmes d'information (CIO[19]).

   Le directeur des systèmes d'information travaille avec les architectes logiciels, les analystes business et les développeurs pour s'assurer qu'une solution s'intégrera correctement avec les systèmes existants, les applications et l'infrastructure.

2.2.1.2 Consommateurs de système

9. Chefs de service de la sécurité (CSO[20]).

[19] Chief Information Officer
[20] Chief Security Officer

Le chef de service de la sécurité travaille également avec les architectes logiciels, les analystes business et les développeurs pour s'assurer que toutes les politiques liées à la sécurité de l'information sont respectées.

10. Directeurs commerciaux.

    Leur mission principale consiste à s'assurer que l'application supporte les objectifs business de l'organisation et que les architectes comprennent tous les enjeux liés à la réglementation.

11. Analystes/clients d'affaires.

    Leurs responsabilités primaires sont d'acquérir et de transmettre aux développeurs la connaissance du domaine business et des exigences fonctionnelles et non fonctionnelles du système.

12. Utilisateurs finaux.

    Leur rôle principal consiste à apprendre à utiliser le système, à préparer et saisir les données à entrer (inputs) et d'interpréter les sorties (résultats) du système. Ils génèrent également certaines exigences système.

13. Développeurs des utilisateurs de services.

    Si le système propose des services disponibles aux clients externes, les architectes ou développeurs qui sont responsables de ces utilisateurs doivent également participer à l'évaluation. Ces développeurs externes peuvent fournir les données en entrées de l'API (A*pplication Program Interface*), ils peuvent aussi participer à l'élaboration du design, et fournir des renseignements sur la qualité de service désirée.

14. Développeurs chargés de maintenance.

    Ils sont responsables des tâches liées à la maintenance générale; seulement, ils ne sont pas capables de modifier les services et modifient par défaut d'autres parties du système.

2.2.1.3 Fournisseurs d'infrastructure.

15. Administrateurs système.

    Leurs responsabilités incluent la bonne compréhension du système pour venir à bout des problèmes qui surgissent pendant et après le déploiement. Ils se chargent habituellement de la plupart des tâches associées à la sécurité de l'ordinateur dans une organisation (entretien du pare-feu et des systèmes de détection d'intrusions, gestion des droits d'accès, et application des patchs du logiciel et du système d'exploitation).

16. Administrateurs réseau.

    L'administrateur réseau maintient l'infrastructure réseau et la résolution des problèmes rencontrés avec la mise en place de routeurs, de switchs et d'ordinateurs sur le réseau.

17. Administrateurs de base de données.

    Ils créent et maintiennent la base de données, et s'assurent de l'intégrité des données ainsi que de la performance des systèmes de gestion de la base de données.

18. Développeurs externes des prestataires de service.

    Si le système accède à des services externes, les architectes ou développeurs qui sont responsables de ces services externes participent également à l'évaluation. Ils peuvent contribuer en spécifiant les exigences requises pour l'interaction avec leurs services, ou même informer sur les points forts et les limites de leurs systèmes

Lorsque nous avons identifié la liste des différents intervenants techniques qui travaillaient sur l'évaluation d'une architecture logicielle, il nous a semblé primordial de connaitre leur degré précis d'implication dans une telle entreprise. Nous avons ainsi décidé de réaliser la matrice du tableau 2-1 qui consiste à croiser le rôle de chacun des intervenants identifiés, avec les attributs qualité du modèle de la norme ISO/IEC 9126-1 que nous pouvons retrouver sur la figure 1-11.

Tab. 2-1 - Matrice Attributs qualité, Intervenants technique

		PRODUCTEURS DU SYSTEME								CONSOMMATEURS DU SYSTEME						FOURNISSEURS D'INFRASTRUCTURE			
	INTERVENANTS LORS DE L'EVALUATION * ► *(voir liste intervenants)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	CRITERES QUALITE ▼																		
CAPACITE FONCTIONNELLE	Pertinence	x	x	x				x		x	x	x							
	Exactitude	x	x	x				x		x	x	x							
	Interopérabilité	x	x			x	x	x	x		x	x		x					x
	Sécurité	x	x					x		x	x					x	x	x	
	Conformité	x	x	x				x			x								
FIABILITE	Maturité	x		x	x		x									x	x	x	
	Tolérance aux pannes	x		x	x		x									x	x	x	
	Facilité de récupération	x		x	x		x									x	x	x	
	Conformité	x		x	x		x									x	x	x	
FACILITE D'UTILISATION	Facilité de compréhension				x								x						
	Facilité d'apprentissage				x								x						
	Facilité d'exploitation				x								x						
	Pouvoir d'attraction				x								x						
	Conformité				x								x						
RENDEMENT EFFICACITE	Comportement temporel	x						x			x					x	x	x	
	Utilisation des Ressources	x						x			x					x	x	x	
	Conformité	x						x			x					x	x	x	
MAINTENABILITE	Facilité d'analyse		x				x								x				
	Modificabilité		x				x							x	x				x
	Stabilité		x				x								x				
	Testabilité		x		x		x								x				
	Conformité		x				x								x				
PORTABILITE	Facilité d'adaptation	x				x	x		x					x					x
	Facilité d'installation	x				x	x		x					x					x
	Coexistence	x				x	x		x					x					x
	Interchangeabilité	x				x	x		x					x					x
	Conformité	x				x	x		x					x					x

En ce qui concerne notre méthode SOAQE que nous détaillerons précisément à partir de la section 3.1, nous avons également besoin de l'intervention de ces mêmes intervenants experts afin qu'ils définissent les points de vues qualité de l'arborescence et qu'ils déterminent les coefficients de pondération des facteurs et critères qualité en jeu (voir section 3.3).

2.3 Les méthodes actuelles d'évaluation

Dans toute évaluation d'architecture logicielle, deux activités sont essentielles pour réussir :

1. Spécifier les attributs qualité qui dérivent des besoins commerciaux.

   Nous avons travaillé avec un ensemble fini d'attributs techniques qui ont un impact important sur la qualité globale, depuis le processus de développement jusqu'au système produit [HkO10b] que nous présenterons dans la section 4.1.1.

2. Sélectionner une assemblée d'intervenants représentatifs pour l'évaluation.

   Durant notre thèse, nous avons travaillé avec une assemblée d'intervenants techniques que nous avons présentés dans la section 2.2.1.

3. Décrire de manière claire et compréhensible l'architecture.

   Notre thèse ne traite pas de cette partie de la problématique que l'équipe a déjà traité [OS08] car elle s'éloignait trop de notre sujet de recherche.

D'une manière générale, les méthodes actuelles d'évaluation arrêtent la décomposition des attributs qualité à l'étape "critère qualité" et demeurent ainsi trop vagues quand il s'agit de donner des mesures précises de la qualité de l'architecture. Ces méthodes ne sont pas précises dans la mesure où elles ne peuvent pas aller plus

loin dans la décomposition de l'architecture et par conséquent elles ne peuvent pas être automatisées au point de définir une valeur finie pour chaque attribut. Elles mènent inévitablement à l'établissement d'un 'brainstorming" entre les intervenants que nous avons listés dans la section précédente (voir section 2.2.1) au bout duquel un arbre de décision est construit (SEI, 2010) (voir figure 2-1).

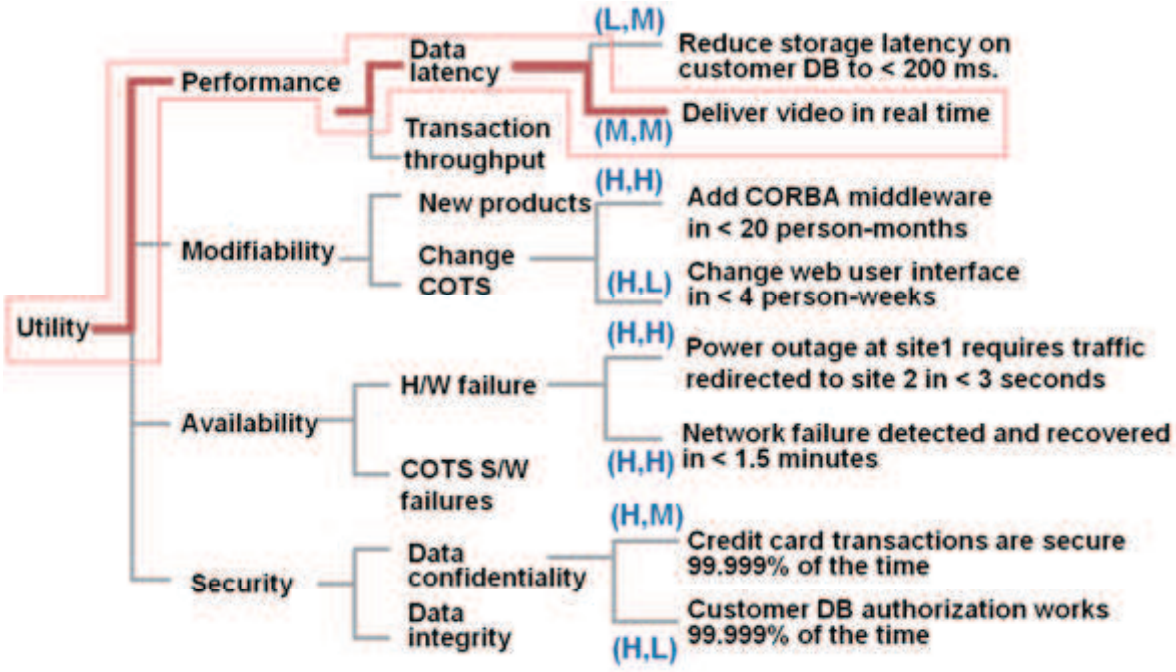

Fig. 2-1 - Arbre d'utilité générateur de scénarii de tests [adaptée depuis KKC00]

Puisque les intervenants n'ont pas les outils permettant la mesure quantitative de chacun des facteurs qualité choisis, ils mettent en place des scénarios visant à respectivement, solliciter séparément chaque facteur, puis critère qualité. Une évaluation très approximative et ad hoc de l'architecture est alors réalisée après avoir étudié le comportement et les réponses du système lors de l'exécution des scénarios de tests.

Par exemple, pour savoir si en termes de performance du système, le temps de latence est géré comme il se doit, nous allons vérifier à travers l'arbre d'utilité si les vidéos sont délivrées en temps réel (voir branche rouge de la figure 2-1); les cardinalités "M,M" signifient que pour une problématique d'importance "moyenne", la réponse du système est évaluée comme "moyenne" ("L, M et H" signifient

respectivement "*Low, Medium* et *High*"). En ce qui concerne les résultats des évaluations avec les méthodes les plus connues ; il existe trois documents communs à toutes les méthodes qui sont systématiquement générés. Ceux-ci retranscrivent une liste hiérarchisée d'attributs qualité en fonction de l'importance de leur considération dans l'architecture soumise à l'évaluation ; une cartographie des approches liées aux attributs qualité de l'architecture et une liste de risques et "non risques".

À présent, rentrons plus en détails sur les principales méthodes existantes.

2.3.1 Taxonomie des méthodes d'analyse et d'évaluation des Architectures logicielles

Nous présentons dans cette section, un ensemble non exhaustif de méthodes d'évaluation de la qualité de l'architecture logicielle les plus référencés. Ainsi, nous décrivons pour chacune de ces méthodes leur principe de fonctionnement, les critères qualité considérés, leurs points forts et leurs limites.

2.3.1.2 La méthode GQM

Nous nous sommes d'abord penchés sur la méthode GQM (pour *Goal/Question/Metrics*) [BCR94] qui est la première méthode d'évaluation majeure que la communauté logicielle a développée. Elle consiste en trois étapes :

1. Définir l'objectif de l'évaluation d'une telle architecture.
2. Définir un ensemble approprié de questions.
3. Associer une métrique à chaque question.

Les limites d'une telle méthode sont apparues rapidement : le fait que le processus ne puisse pas être automatisé parce que les différents objectifs (goals) visés (et donc des questions/métriques résultant de ces objectifs) sont exclusivement définis par des intervenants humains fausse inévitablement les résultats car les intervenants, aussi techniques soient-ils, ne peuvent pas couvrir systématiquement toutes les exigences du système pour évaluer la qualité.

Nous avons alors vu des méthodes assez ressemblantes voir le jour, telles qu'ATAM ou SAAM [CKK02], qui ont propulsé l'évaluation de l'architecture logicielle comme étant une étape standard de tout paradigme.

2.3.1.3 Architecture tradeoff analysis method (ATAM)

ATAM est la méthode d'évaluation qui est celle étant la plus largement utilisée lors de l'évaluation d'architectures. L'aptitude qu'a une architecture à répondre favorablement aux objectifs attendus est déterminée par les exigences qualité qui sont très importantes pour les intervenants du système [DN02b]. La méthode ATAM repose, comme nous l'avons présenté dans la section 2, sur la résolution de scénarios relatifs aux attributs qualité tels que: la performance, la fiabilité, la disponibilité, la sécurité, la modificabilité, la portabilité, la fonctionnalité, la variabilité, l'adaptabilité, l'intégrité conceptuelle. Cette méthode a été créée pour découvrir les risques et les compromis engendrés par les décisions architecturales liées à ces attributs qualité. Elle permet de simuler des scénarios d'utilisation du logiciel; et chacun de ces scénarios isole un unique attribut qualité afin que l'on vérifie si ce dernier est positivement traité par l'architecture actuelle ou non. Ces scénarios nous donnent des réponses précises aux conditions d'utilisation, de performance et d'évolution de l'architecture. Ils nous permettent également de découvrir les divers types de failles ainsi que les menaces éventuelles. Une fois que les attributs qualité les plus importants sont identifiés, les décisions architecturales liées à chaque scénario de haute priorité sont analysées. En effet, outre l'évaluation des attributs qualité, ATAM explore également l'interaction des attributs qualité et leurs interdépendances. Le but de cette analyse réalisée en neuf étapes est d'identifier les risques, les "non risques", les mécanismes de compromis et les points sensibles de l'architecture. Un rapport final des résultats ATAM renferme, en plus des résultats communs à toutes les méthodes non automatisées, un résumé des approches architecturales utilisées, une liste des points sensibles et des "compromis", l'analyse de chaque scénario choisi accompagné de ses attributs qualité correspondants et d'importantes conclusions [KKC00].

La figure 2-2 récapitule les différentes phases de la méthode ATAM et nous permet de comprendre que les éléments business essentiels et l'architecture logicielle sont définis par les décisionnaires du projet. Ils sont ensuite traduits en scenarios que l'on combine aux décisions architecturales du projet. L'analyse des scénarios et des décisions a comme conséquence l'identification des risques, des non-risques, des points sensibles et des compromis de l'architecture. Les risques sont synthétisés en un ensemble fini de thèmes de risques, en montrant comment chacun menace potentiellement une fonctionnalité majeure du système.

Fig. 2-2 - Les phases d'ATAM (adaptée à partir de [KKB+98])

2.3.1.4 Software Architecture Analysis Method (SAAM)

Cette méthode que l'on peut considérer comme le précurseur d'ATAM, a également été initiée par l'institut Carnegie Mellon[21]. Par contraste à ATAM, les auteurs de cette méthode cherchaient une méthode capable de se concentrer sur la *modifiabilité* dans ses diverses formes (telles que la portabilité, la variabilité…) et la fonctionnalité. Toutefois, le mécanisme est identique, à savoir, une évaluation menée

[21] Université de Carnegie Mellon, Etats-Unis

par le biais de scénarios imaginés afin de vérifier si l'architecture peut faire face aux dépenses réelles [KBA+94, Mol99].

Si une seule architecture est analysée, SAAM indique ses points forts et ses points faibles, et souligne les points où l'architecture ne répond pas aux besoins liés à l'adaptabilité.

Si deux ou plusieurs architectures sont comparées en fonction de leur capacité à répondre favorablement à l'attribut qualité "adaptabilité", SAAM est capable d'effectuer un classement relatif entre elles [IHO02].

Le principe de fonctionnement de la méthode SAAM est schématisé dans la figure 2-3

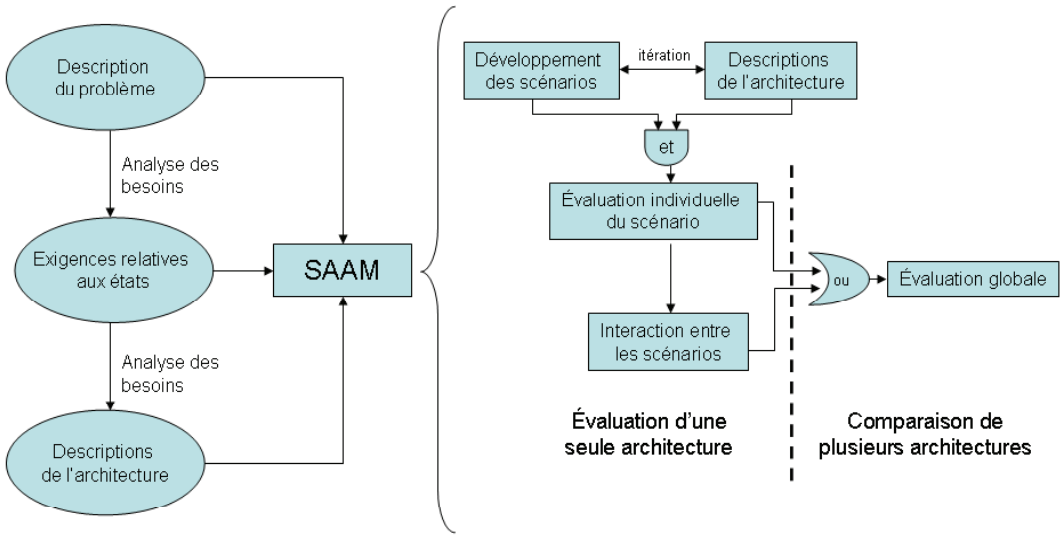

Fig. 2-3 - Principe de fonctionnement du SAAM (adaptée à partir de [DN02a])

La figure 2-3 nous montre assez clairement qu'avant de vouloir évaluer son architecture à l'aide de SAAM, nous devons d'abord être en mesure de décrire les problèmes et l'architecture et de lister les exigences du système. Tout comme ATAM, il existe pour SAAM une phase dédiée à l'identification de scénarios de test pour pouvoir évaluer une ou plusieurs architectures.

2.3.1.5 SAAM Founded on Complex Scenarios (SAAMCS)

SAAMCS est une méthode d'évaluation étant basée sur, et censée améliorer, la méthode SAAM. Cette méthode considère que la complexité de scénarios est le facteur le plus important pour l'évaluation des risques. [LRV99].

Les contributions de SAAMCS, additionnellement à celles de SAAM, consistent essentiellement à savoir identifier les scénarios parfois complexes à réaliser et d'autre part, à évaluer où leur impact est évalué. Une liste de scénarios difficiles à mettre en œuvre est établie en se basant sur l'initiateur du scénario, la description de l'architecture logicielle et la version des conflits.

Par ailleurs, l'évaluation des risques constitue le but le plus recherché pour la méthode SAAMCS. Elle octroie un rôle très important à l'initiateur du scénario car il représente l'unité organisationnelle qui a le plus d'intérêts dans la mise en œuvre de ce scénario.

SAAMCS est appliquée à la version finale de l'architecture, qui doit être décrite de manière suffisamment détaillée comme nous pouvons le voir sur la gauche de la figure 2-4.

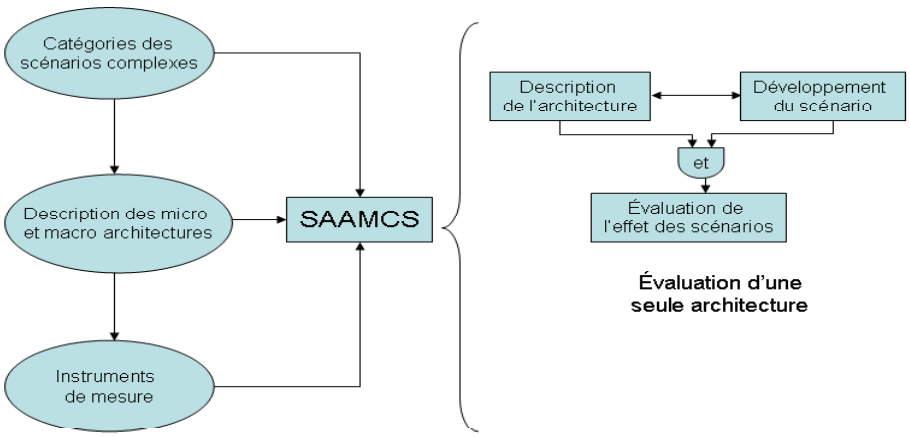

Fig. 2-4 - Description de la méthode SAAMCS (adaptée à partir de [DN02a])

2.3.1.6 Cost-Benefit Analysis Method (CBAM)

CBAM commence là où ATAM s'achève; dans la mesure où c'est une méthode centrée sur l'architecture pour l'analyse des coûts, les bénéfices et l'ordonnancement des implications des décisions architecturales [CBA02]

CBAM évalue également le degré d'incertitude associé à ces décisions, en fournissant une base pour le processus de décision concernant l'architecture [NBC+03].

En outre, CBAM est considérée comme une passerelle entre deux domaines dans le génie logiciel, à savoir l'architecture logicielle et l'économie d'organisation. CBAM permet d'ajouter les coûts (budgets implicites) comme critères de qualité, qui doivent être considérés lorsqu'un logiciel est planifié et prétend que les coûts, les bénéfices et les risques sont aussi importants que les autres attributs qualité [DN02b].

2.3.1.7 Architecture-Level Modifiability Analysis (ALMA)

La méthode ALMA a été développée et testée pour Business Information Systems (BIS), comme étant une méthode fondée sur des scénarios pour l'évaluation des attributs qualité architecturaux d'un logiciel, en mettant spécifiquement l'accent sur la *modifiabilité*. ALMA peut également être applicable pour les *systèmes embarqués* [BLB+04, Las02]

En général, l'analyse de la *modifiabilité* a pour objectifs les points suivants :

- Prévision des coûts des futures modifications.
- Identification de l'inflexibilité du système.
- Comparaison de deux ou plusieurs architectures alternatives.

En outre, ALMA est recommandée pour l'analyse du critère de *modifiabilité* en utilisant une série d'indicateurs : la prédiction des coûts de maintenance et l'évaluation du risque. ALMA est capable d'évaluer et de comparer plusieurs systèmes et cette méthode utilise également des scénarios de changements fournis par les intervenants du système : l'analyse de la *modifiabilité* commence par la

définition d'un ensemble de scénarios qui pourraient se produire lors de l'exécution du système. [IHO02, BLB+04].

2.3.1.8 Family-Architecture Analysis Method (FAAM)

La méthode FAAM est principalement conçue pour l'évaluation de l'architecture des *systèmes d'information*. Elle se concentre sur deux aspects de la qualité logicielle qui sont : l'*interopérabilité* et l'*extensibilité* [Dol02].

L'objectif de FAAM est d'établir un processus pour l'évaluation de l'architecture des systèmes d'information. Ce processus considère les lignes directrices, les métriques et les recommandations. En outre, FAAM est différente des autres méthodes d'analyse architecturale et contribue à :

- Associer activement les parties prenantes du produit dans le processus de création de produits.
- Mettre l'accent sur les critères d*'interopérabilité* et d'*extensibilité* des systèmes d'information.
- Souligner les mécanismes et les techniques du savoir-faire pour permettre aux équipes de développement de mettre en œuvre la méthode.

2.3.2 Comparaison des différentes méthodes d'évaluation logicielle

Afin de comparer les différentes méthodes d'analyse, nous avons défini un ensemble de critères de comparaison (voir Tableau 2-2). Quelques uns de ces critères sont déjà présentés dans [DN02b].

Tab. 2-2 - Les critères d'évaluation d'une méthode d'évaluation

Critère	**Description**
C1 : Objectif spécifique	Quel est l'objectif spécifique de la méthode ?
C2 : Les attributs de qualité	Quels sont les attributs qualité considérés ?
C3 : Participation des parties prenantes	Quelles sont les parties prenantes qui participent au processus d'évaluation ?
C4 : Réutilisation d'un savoir existant	Si une base de connaissance est considérée, comment est-elle structurée pour une réutilisation efficace ?
C5 : Validation de la méthode	La méthode est-elle validée en pratique ?
C6 : Points forts	Quels sont les points forts de la méthode ?
C7 : Limites	Quelles sont les limites de la méthode ?
C8 : Systèmes	Quels sont les systèmes sur les lesquels la méthode peut être appliquée ?

Nous les complétons et les argumentants différemment dans cette section puis établissons une réponse à ces critères pour chacune des méthodes précédemment présentées.

Tab. 2-3 - Les critères appliqués aux méthodes d'évaluation logicielle ATAM, SAAM et SAAMCS

Méthode	**ATAM**	**SAAM**	**SAAMCS**
C1	Scénarios, Intègre les techniques de mesures	Scénarios	Scénarios
C2	performance, fiabilité, disponibilité, sécurité, modificabilité, portabilité, fonctionnalité, variabilité, adaptabilité, l'intégrité conceptuelle	*Modifiabilité* (portabilité, variabilité), fonctionnalité, adaptabilité	Flexibilité
C3	Tous ou seul le concepteur	Tous	Tous
C4	Un ensemble de paquets d'analyses et de question avec des solutions connues	Aucune	Aucune
C5	Oui	Oui	Oui
C6	Génération de scénarios en fonction des besoins Applicable pour les propriétés statiques et dynamiques arbre d'utilité de la qualité	Identification des zones de complexité potentielle Ouvert à toute description architecturale	Identification des zones de complexité potentielle Ouvert à toute description architecturale
C7	Nécessite des connaissances techniques approfondies	Critère de qualité pas clair N'est pas supportée par des techniques pour réaliser les	Critère de qualité pas clair N'est pas supportée par des

		étapes	techniques pour réaliser les étapes
C8	Tous	Tous	Tous

Tab. 2-4 - Les critères appliqués aux méthodes d'évaluation logicielle CBAM, ALMA ET FAAM

Méthode	**CBAM**	**ALMA**	**FAAM**
C1	Intègre les techniques de mesure	Scénarios	Tables et diagrammes spécifiques
C2	Coûts, bénéfices, ordonnancement des implications	Modificabilité	Interopérabilité et extensibilité
C3	Tous	Tous	Tous
C4	Aucune	Aucune	Aucune
C5	Oui	Oui	Oui
C6	Prévoir des mesures économiques pour les changements spécifiques du système Rendre explicite l'incertitude associée aux estimations	Critère d'arrêt pour la génération des scénarios	Insiste sur la responsabilisation des équipes dans l'application de la session FAAM
C7	Identification et estimation des coûts et bénéfices pouvant être faits par les participants d'une manière ouverte	Ensemble restreint d'études de cas Concentration	Partiellement prouvée dans un environnement particulier Concentration sur les propriétés statiques

		sur les propriétés statiques	
C8	Tous	Systèmes d'information commerciaux	Systèmes d'information dits produits familiers

2.3.3 Conclusion sur les méthodes d'évaluation

Concrètement, avec les méthodes non automatisées actuelles, les évaluations des SOA produisent toutes un rapport dont la forme et le contenu varient en fonction de la méthode employée.

Mais, d'une façon générale, l'évaluation produit de l'information textuelle et répond à deux types de questions [CKK01] :

-1- Est-ce que cette architecture est adaptée au système pour lequel elle a été conçue ?
-2- Y a-t-il une autre architecture davantage adaptée au système en question ?

-1- Nous pouvons dire que l'architecture est adaptée si elle répond favorablement aux trois points suivants:

i. Le système est prévisible et peut répondre aux exigences qualité et aux contraintes de sécurité des spécifications.

ii. Pas toutes les propriétés qualité du résultat du système dérivent directement de l'architecture mais la plupart en sont issues; et pour ces dernières, l'architecture est considérée appropriée s'il est possible d'instancier le modèle en prenant en considération ces propriétés.

iii. Le système peut être mis en place en utilisant les ressources actuelles : le personnel, le budget, et le temps alloué avant la production du résultat. En d'autres termes, l'architecture est réalisable.

Cette définition ouvrira la voie aux futurs systèmes et a évidemment des conséquences majeures. Si les intervenants d'un système ne peuvent pas nous dire quels sont les attributs à contrôler en priorité, et bien n'importe quelle architecture pourra nous renseigner [CKK01].

-2- Une partie de l'évaluation des SOA consiste à capturer les attributs qualité que l'architecture doit manipuler, puis à privilégier le contrôle de ceux-ci. Si la liste des attributs qualité convient dans le sens où tous les objectifs commerciaux sont, au moins, indirectement considérés, alors, nous pouvons continuer à travailler avec la même architecture. Dans le cas contraire, il est nécessaire de travailler avec une architecture plus appropriée au système et pour ce, s'en référer éventuellement à la solution SACAM (voir section Partie 1, SACAM).

Ces attributs qualité peuvent être en conflit pour atteindre certains des objectifs commerciaux. En pareil cas, le chef de projet doit prendre la décision de se concentrer sur un ensemble limité d'attributs (c'est précisément ce que nous avons décidé de faire pour notre modèle que nous découvrirons dans la section 4), d'autant plus si l'évaluation de l'architecture donne un résultat concluant dans un secteur et un résultat moins cohérent dans un autre secteur [LMB07].

Cependant, à condition qu'il soit toujours possible de répondre favorablement aux exigences non fonctionnelles (attributs qualité) et fonctionnelles des spécifications, l'architecture choisie peut être maintenue malgré les défaillances relevées.

De plus, puisque chaque secteur est lié à une liste d'objectifs et que ceux-ci sont atteints en se concentrant sur certains attributs qualité, la meilleure manière de renforcer les secteurs négligés est de produire un travail plus robuste sur les attributs concernés par le secteur en question. L'évaluation aidera à indiquer où l'architecture engendre des défaillances, mais l'équilibre entre le coût de l'évaluation et l'aide qu'elle fournit pour améliorer le projet reste relatif à toute architecture.

Finalement, l'évaluation de l'architecture ne répondra pas aux questions quant à savoir si l'architecture est adaptée ou non; si elle est "bonne" ou "mauvaise" ou si on lui attribut la note de "sept sur dix"; l'évaluation nous indique seulement où réside le danger. Elle peut être appliquée à une unique SOA ou à un groupe de plusieurs SOA candidates.

Dans ce dernier cas, l'évaluation identifie d'abord les objectifs commerciaux majeurs requis pour la comparaison et puis, elle examine la documentation disponible pour chaque architecture candidate. Finalement, elle évalue l'architecture choisie, récapitule les résultats d'analyse et fournit une recommandation pour le processus décisionnel.

2.3.4 Discussion sur les méthodes d'évaluation

L'évaluation des SOA se rapporte à des approches qualitatives et quantitatives et à des prévisions de charges associées à des évolutions et des limites théoriques d'une architecture donnée.

Pendant l'évaluation des architectures logicielles, nous pesons la pertinence de chacune des problématiques associées à la phase de conception après avoir évalué l'importance de chaque exigence qualité.

Dans cette perspective, les outils et les approches existants ont montré leurs limites pour les SOA [LMB07] : nous avons longtemps observé ces dernières décennies la naissance d'une nouvelle génération d'outils non automatisés, développés par les industriels (ATAM, SAAM [CKK01]) mais les résultats obtenus n'ont pas été considérés comme satisfaisants : ils sont souvent très différents et aucune de ces dernières n'y pallie parfaitement. Non seulement de nombreuses inquiétudes majeures ont été soulevées avec ces méthodes [BKM07] ; en particulier leur coût en termes de temps (il existe beaucoup d'étapes à effectuer pour finaliser le processus d'évaluation) et d'argent parce que les évaluations effectuées sont exclusivement manuelles ; mais en plus ; la défaillance principale de cette génération de méthodes concerne la pauvreté des résultats des évaluations conduites. Plus précisément, les attributs qualité forment la base pour l'évaluation architecturale de la qualité, mais

nommer sommairement des attributs qualité n'est pas une base suffisante sur laquelle juger de la robustesse d'une architecture.

Souvent, les exigences qualité de la spécification systèmes sont affichées comme telles :

- Le système doit être hautement modifiable.
- Le système doit être sécurisé.
- Le système doit afficher des performances acceptables…

Sans élaboration approfondie, chacune de ces exigences qualité peut être sujette à une incompréhension ou un malentendu. Ce qu'un intervenant peut considérer comme robuste, un utilisateur peut le considérer inadéquat et vice-versa.

Peut-être qu'un système peut facilement adopter une nouvelle base de données mais pas un nouveau système d'exploitation. Est-ce que ce système est maintenable ou non ?

Peut-être qu'un système utilise des mots de passe en guise de sécurité mais n'a aucun mécanisme de protection contre les virus. Est-ce que ce système est sécurisé ou non ?

La conclusion de ces problématiques consiste à dire que les attributs qualité ne sont pas des mesures quantitatives absolues.

L'ampleur de la tâche a amené le monde académique à se pencher sur ces problématiques afin d'essayer de développer une approche éventuellement automatisable mais surtout plus formelle et générique que les méthodes que nous venons de lister.

De nouveaux efforts ont été entrepris pour évaluer les SOA dans divers aspects et en utilisant des outils différents, notamment des méthodes comme [MW08] où la solution des "attack graphs" est adoptée pour les métriques de sécurité d'une SOA (*Les administrateurs système analysent les graphes d'attaque pour comprendre où les défaillances de leur système réside et pour décider quelles mesures liées à la*

sécurité seront prêtes à être déployées). Mais la majorité de ces recherches ne sont que des propositions ou utilisent des techniques très différentes ou encore, ne concernent que quelques aspects de l'évaluation logicielle [CDR03].

C'est à ce niveau que notre travail diffère de ceux existants dans la mesure où nous souhaitons obtenir une mesure précise pour chaque attribut qualité de notre arborescence. Spécifiquement, nous voulons semi-automatiser le processus dans le but d'éviter des évaluations exclusivement manuelles incitant à systématiquement solliciter les intervenants pour toutes les phases de l'évaluation. Ceux-ci déterminent comme points de départ de l'évaluation, des attributs qualité à exploiter sans être sûrs que tous les aspects architecturaux soient couverts. Avant de passer à notre solution en détail et après avoir profondément étudié les méthodes actuelles d'évaluation de la qualité d'une architecture, nous nous penchons désormais assez brièvement sur le cas de l'évaluation logicielle à l'aide d'outils logiciels. Pour ce, nous dressons une liste non exhaustive d'outils logiciels existants pour évaluer quantitativement, cette fois ci, et à l'aide de métriques logicielles connues, la qualité d'une architecture à partir de son code source.

2.4 Outils de mesure qualité

Il existe une large panoplie de métriques qualité logicielles qui ont été développées, et de nombreux outils existent pour collecter les métriques depuis les représentations de programmes. Cette large variété d'outils permet à l'utilisateur de sélectionner celui qui lui correspond le mieux, en fonction par exemple du prix de l'outil, du support outil ou de sa capacité à le manier.

Définitions : Nous avons considéré pour l'ensemble de nos travaux qu'une métrique est une définition mathématique qui lie une entité quantifiable d'un système logiciel à des valeurs métriques. Aussi, nous appelons un outil de métriques logicielles, un programme qui implémente un ensemble de définitions de métriques logicielles. Cet outil permet d'évaluer le système logiciel en fonction des métriques en extrayant les

entités requises depuis le logiciel et qui fournissent les valeurs des métriques correspondantes. Le modèle "Facteurs Critères Métriques" de McCall (voir section 2), qui nous a servi de base pour l'ensemble de nos travaux et qui s'applique au logiciel mène à la notion de modèle de qualité logicielle. Ce modèle combine des valeurs de métriques logicielles à des valeurs numériques agrégées dans le but précis d'évaluer et d'analyser la qualité d'une architecture. Le modèle de McCall a d'ailleurs, comme nous l'avons précédemment montré dans la section 2, également servi de base au modèle de qualité logicielle ISO 25000 SQuaRE (anciennement ISO/IEC 9126-1).

L'objectif majeur de cette section du manuscrit consiste à savoir si ce nouveau procédé d'évaluation de la qualité logicielle par le biais d'outils de métriques logicielles répond aux problèmes que connaissent les méthodes d'évaluation de la qualité logicielle que nous avons étudiés dans la section 3; à savoir, obtient-on avec ces outils des conclusions cohérentes, pertinentes et semblables quel que soit l'outil de mesure utilisé en guise de résultats de l'évaluation ?

Pour ce :

1. Nous sélectionnons une liste non exhaustive d'outils de métriques logicielles.
2. Nous sélectionnons une liste de métriques logicielles.
3. Nous lançons les processus de calcul de ces métriques par le biais de ces outils.
4. Nous comparons les résultats obtenus.

Idéalement, nous aimerions considérer l'ensemble des outils existants et avec ceux-ci, mesurer l'ensemble des métriques logicielles qui ont été répertoriées. Mais évidemment, chaque outil ne permet pas de mesurer toutes les métriques existantes. Il existe en effet un certain nombre de limites pratiques. Corrélativement, nous ne pourrons pas considérer toutes les métriques logicielles car cela exclurait tous les outils, or le but de cette étude consiste à les comparer entre eux.

L'idée consiste alors à faire des compromis et à sélectionner les métriques logicielles qui nous sont intéressantes dans le cadre de notre travail. La majorité d'entre elles

proviennent des suites de métriques logicielles connues (Chidamber & Kemerer, Li et Henry, McCabe, Halstead etc.) [Gor06, Hal77, Hen96, KKL+01].

Aussi, il existe beaucoup d'outils de métriques logicielles et ils ne sont pas tous faciles à trouver. La sélection des outils s'est exclusivement basée sur ceux qui étaient disponibles à partir de recherches standards sur Internet, sans restrictions légales, et en parcourant les articles que l'on a référencés dans nos travaux.

Nous avons pu d'abord répertorier une cinquantaine d'outils logiciels capables de mesurer les métriques logicielles mais au vu des limites que nous nous sommes fixés, à savoir, travailler exclusivement avec des outils freeware capables d'extraire les métriques logicielles courantes à partir de programmes JAVA dans leur version complète et non d'évaluation ; il ne nous restait finalement plus que six outils de métriques logicielles, à savoir :

1. **CCCC**[22] est un outil open source en lignes de commande. Il analyse du code C++ et Java et génère des rapports sur plusieurs métriques, notamment LOC (pour *Lines of Code*) et les métriques proposées par Chidamber & Kemerer et Henry & Kafura**.**
2. **Chidamber & Kemerer Java Metrics**[23] est un outil open source en lignes de commande. Il calcule les métriques C&K orientées objet en traitant le code-octet du code Java compilé
3. **Dependency Finder**[24] est un open source en lignes de commande. C'est une suite d'outils pour l'analyse du code Java compile. Son cœur est une application d'analyse des dépendances qui extrait des graphes de dépendances pour y soutirer des informations utiles. C'est une application "Swing based", une application web et un ensemble de tâches "Ant".
4. **OOMeter** est un outil experimental de métriques logicielles développé par Alghamdi et al. Il prend en entrée du code source Java ou C#, et des modèles UML en XMI et calcule diverses métriques [ARK05, Als03]

---

[22] *C and C++ Code Counter* – http://www.cccc.sourceforge.net

[23] http://www.spinellis.gr/sw/ckjm/

[24] http://www.depfind.sourceforge.net

5. **Semmle**[25] est un plug-in Eclipse. Il fournit une sorte de langage de requêtes SQL pour du code orienté objet, ce qui permet de rechercher les bugs ; mesurer des métriques etc.
6. **Understand for Java**[26] est un outil permettant d'explorer du code Java.

Nous avons ensuite tenté d'établir une liste complète des métriques logicielles existantes. Nous en avons compté plus d'une centaine ayant des appellations différentes mais en réalité beaucoup se ressemblent dans la mesure où elles font le même travail. Finalement nous avons réduit notre liste afin de n'en garder plus qu'une quarantaine. Afin que l'on puisse comparer concrètement ces métriques logicielles, nous avons pris le soin de ne sélectionner que les métriques qui ne sont calculées qu'au niveau des classes du code ; la plupart des métriques ont été établies pour ce niveau d'abstraction précisément ; il ne nous en restait plus qu'une quinzaine. Et finalement nous avons voulu travailler avec des métriques qui étaient majoritairement traitées par les outils que l'on a sélectionnés. Finalement il ne nous en restait plus que sept. Voici une très brève définition des métriques gardées (*nous reviendrons beaucoup plus en détails sur les métriques concernant directement nos travaux, notamment celles qui concernent le couplage dans la section 3*) :

1. **CBO** (Coupling Between Object classes) est le nombre de classes auxquelles une classe est couplée [CK91].
2. **DIT** (Depth of Inheritance Tree) est le chemin maximal d'héritage depuis une classe jusqu'à la classe racine [Gor06].
3. **LCOM-CK** (Lack of Cohesion of Methods) (Chidamber & Kemerer) décrit le manqué de cohésion parmi les méthodes d'une classe [CDK98].
4. **LOC** (Lines Of Code) compte le nombre de lignes de code d'une classe [Hal77].

---

[25] http://www.semmle.com
[26] http://scitools.com

5. **NOC** (Number Of Children) est le nombre de sous-classes immédiates d'une classe dans la hiérarchie de la classe [Hal77].
6. **NOM** (Number Of Methods) est le nombre de méthodes d'une classe [Hen96].
7. **RFC** (Response For a Class) est l'ensemble des méthodes qui peuvent potentiellement être exécutées en réponse à un message reçu par un objet de la classe [CK94].

Les outils et les métriques sont affichés dans le tableau 2-5. La croix "x" signifie que l'outil en question est en mesure de calculer la métrique correspondante.

Tab. 2-5 - Outils et métriques utilisées pour notre évaluation

Tools	Metrics						
Name	CBO	DIT	LCOM-CK	NOC	NOM	RFC	LOC
CCCC	x	x		x	x		
Chidamber & Kemmerers Java Metrics	x	x	x	x	x	x	
Dependency Finder		x		x	x		x
OOMeter	x	x	x	x			
Semmle		x	x	x	x	x	x
Understand for Java	x	x	x	x	x		x

Nous lançons les évaluations sur un projet que l'on a à BeOtic (et qui nous servira également d'étude de cas pour l'expérimentation de notre solution d'évaluation dans la section 4 du manuscrit) avec les outils que l'on a à notre disposition (certains sont des plug-ins à intégrer à un environnement de développement, d'autres sont des open-source en lignes de commande et d'autres fournissent une interface graphique à l'utilisateur). La plupart des outils ont généré des rapports HTML ou XML contenant un résumé des valeurs des métriques et d'autres résultats d'analyse que l'on a ignorés. Nous avons reporté les résultats dans le tableau 2-6 suivant :

Tab. 2-6 - Résultats d'évaluations et différences entre les outils métriques

Tool	Data	CBO	DIT	LCOM-CK	LOC	NOC	NOM	RFC
C&K Java Metrics	Max of Value	42.000	3.000	795.000		12.000	47.000	170.000
	Min of Value	0.000	0.000	0.000		0.000	2.000	3.000
	Average of Value	2.826	0.478	23.565		0.674	6.087	16.391
CCCC	Max of Value	19.000	2.000			12.000	44.000	
	Min of Value	1.000	0.000			0.000	2.000	
	Average of Value	4.043	0.870			0.674	5.652	
Dependency Finder	Max of Value		2.000		341.000	12.000	47.000	
	Min of Value		1.000		5.000	0.000	2.000	
	Average of Value		1.674		28.130	0.674	6.609	
OOMeter	Max of Value	0.500	2.000	853.000		12.000		
	Min of Value	0.000	1.000	1.000		0.000		
	Average of Value	0.180	1.705	30.727		0.705		
Semmle	Max of Value		4.000	728.000	585.000	12.000	43.000	177.000
	Min of Value		1.000	0.000	6.000	0.000	0.000	0.000
	Average of Value		1.913	21.196	46.804	0.674	5.000	11.130
Understand For Java	Max of Value	62.000	4.000	92.000	874.000	12.000	44.000	
	Min of Value	0.000	1.000	0.000	11.000	0.000	2.000	
	Average of Value	4.500	1.913	39.435	68.587	0.674	5.652	

Pour avoir une meilleure compréhension des résultats nous avons généré des tableaux croisés dynamiques nous affichant les valeurs minimales et maximales obtenues ainsi qu'une moyenne de ces valeurs. La première chose que nous constatons est qu'il existe des différences assez importantes entre les résultats d'évaluation des différents outils pour une même métrique. Par exemple, nous constatons que la moyenne des classes du système pour la métrique LCOM-CK, varie entre 21.196 (Semmle) et 39.435 (Understand for Java). D'un autre côté, nous nous apercevons qu'en ce qui concerne la métrique NOC, les outils calculent tous des valeurs identiques pour ce projet (excepté la valeur de l'outil OOMeter qui est sensiblement plus élevée) ; pareillement pour la métrique NOM.

Notre explication pour les différences constatées entre LCOM-CK et les similitudes de résultats pour NOC ou NOM, réside essentiellement dans le fait que des métriques comme LCOM-CK sont beaucoup plus complexes dans leurs descriptions et qu'il est aisé au vu de la complexité de leurs description d'obtenir des implémentations différentes pour une même métrique. Des métriques comme NOC ou NOM sont très simples à décrire et à implémenter, ce qui explique la similitude des résultats (les différences obtenues pour la métrique LOC qui est également une métrique plutôt claire et simple, à savoir de 28.130 à 68.587 s'expliquent par le fait que chacun des outils compte différemment ses lignes de codes, certains comptent les lignes dans le corps des méthodes uniquement, d'autres toutes les lignes depuis le

début de la déclaration de la classe jusqu'à sa fin en excluant les déclarations de méthodes, soit les lignes effectives exclusivement).

Ce tableau assez éloquent répond par la négative à notre question initiale, quant à savoir si les résultats d'évaluation obtenus avec les outils de métriques logicielles existants étaient plus cohérents et pertinents que ceux obtenus avec les méthodes d'évaluation que l'on connaît et nous incite à considérer et à travailler sur cette problématique à travers notre solution.

2.5 Conclusion

Nous avons présenté à travers ce chapitre, une synthèse des différentes méthodes d'évaluation de la qualité logicielle et un corpus non exhaustif d'outils d'analyse du code. Pour cette étude, nous avons sélectionné les outils les plus populaires dans le marché du logiciel.

Par ailleurs, nous avons pu constater que la majorité presque absolue des outils étudiés concerne l'extraction des métriques et peu de ces outils abordent la notion de qualité d'architecture logicielle.

La multitude d'échecs et d'abandons de projets que connaissent les organisations ces dernières années peut s'expliquer par l'inefficacité des solutions actuelles. Puisque nous avons vu que les méthodes et les outils étaient défaillants, dans la mesure où les méthodes donnent des résultats trop vastes et que les outils donnent des résultats trop éloignés, alors notre contribution consiste à concevoir notre propre application composée d'un triptyque {modèle, méthode, outil} complet pour pallier aux échecs que l'on connait actuellement et pouvant être en mesure d'évaluer la qualité d'une architecture logicielle orientée service. Ce triptyque est introduit à partir de la section 3 suivante.

Chapitre 3

SOAQE : UN MODELE ET UNE MÉTHODE D'ÉVALUATION DE LA QUALITÉ

3.1 Introduction

Comme nous l'avons établi dans le chapitre précédent, les méthodes actuelles d'évaluation arrêtent la décomposition des attributs qualité trop tôt dans le cycle de vie de l'évaluation de la qualité de l'architecture et demeurent ainsi trop vague lorsqu' il s'agit de donner des mesures précises comme résultats de l'évaluation.

C'est précisément à ce moment que notre travail diffère de ceux existants dans la mesure où nous souhaitons obtenir une mesure quantitative précise pour chaque attribut qualité à l'aide du modèle et de la méthode SOAQE.

Nous visons principalement à semi-automatiser le processus afin d'éviter des évaluations exclusivement manuelles poussant à systématiquement à solliciter un tas d'intervenants différents durant tout le processus d'évaluation, depuis la décomposition de l'architecture orientée service en attributs qualité jusqu'à l'obtention des résultats de l'évaluation.

3.2 Le modèle SOAQE

Nous l'avons indiqué dans la section 1.8.1 ; pour concevoir notre modèle prévisionnel de qualité, nous nous sommes fortement inspirés du modèle de McCall. En effet, comme pour ce dernier, nous avons repris la décomposition de nos architectures orientées service en plusieurs niveaux, quatre en l'occurrence, dont trois communs au modèle de McCall (facteurs qualité, critères qualité et métriques qualité) auxquels nous avons ajouté le niveau point de vue qualité.

Il existe évidemment une notion de hiérarchisation dans cette découpe ; comme nous le montre la figure 3-1 ci-après, plus nous nous "enfonçons dans l'entonnoir", plus l'élément est granulaire et sa définition est concrète. Réciproquement plus nous "remontons l'entonnoir" plus l'élément est de nature abstraite.

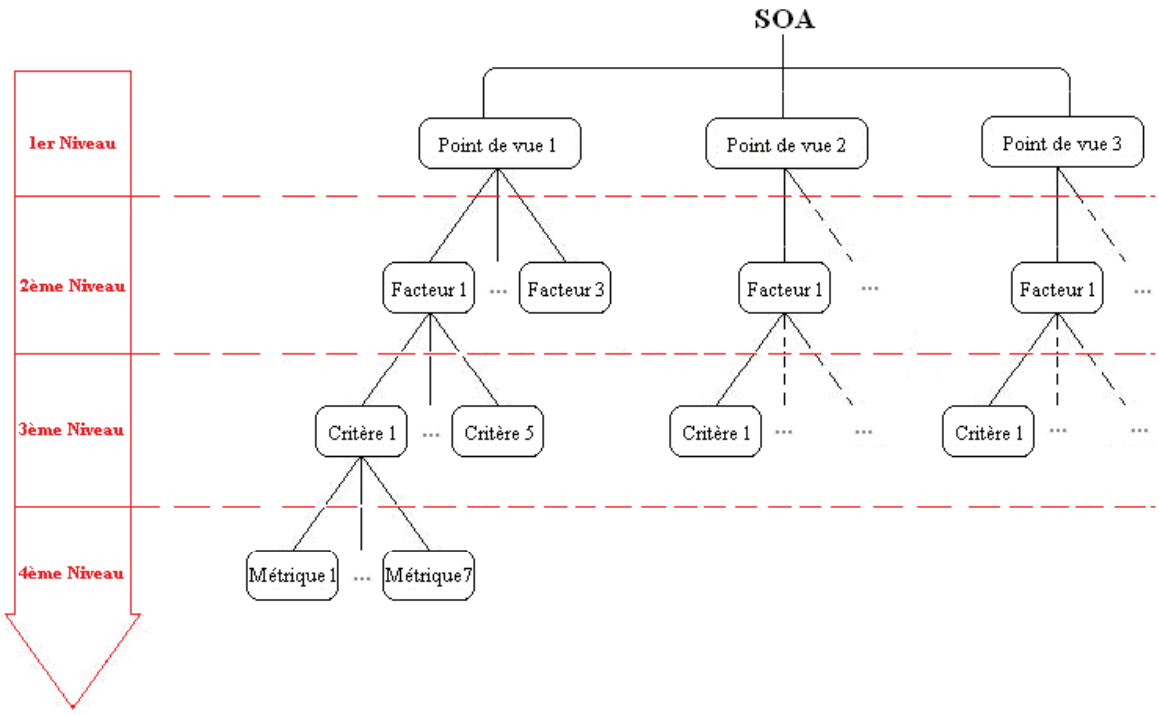

Fig. 3-1 – Cartographie d'une architecture orientée service

3.2.1 Les points de vue qualité

Le premier niveau de décomposition de notre architecture orientée service est le point de vue qualité. Un point de vue qualité permet de regrouper en familles les attributs qualités ayant des caractéristiques semblables ; l'intérêt d'une telle décomposition réside dans le fait que toute architecture orientée service réunira, après décomposition en attributs qualité, les mêmes points de vue qualité, ce qui permet d'automatiser le processus d'évaluation afin de l'étendre à toutes les SOA considérées. Nous étant naturellement positionnés dans le rôle d'architectes depuis le début de la thèse et ayant décidé de travailler exclusivement avec des attributs qualité relevant du domaine du "technique", considérés, parmi ceux recensés, comme étant les plus importants pour la communauté des architectes du génie logiciel, nous nous sommes immédiatement concentrés sur le point de vue qualité **architectural** et n'avons que très récemment entrepris de nouveaux travaux de recherche sur les facteurs, critères et métriques qualité composant le point de vue qualité **business** pour étoffer notre architecture (la figure 3-2 montre une modélisation de notre

architecture sous le point de vue qualité business faite avec notre outil "MindMap"[27] à BeOtic).

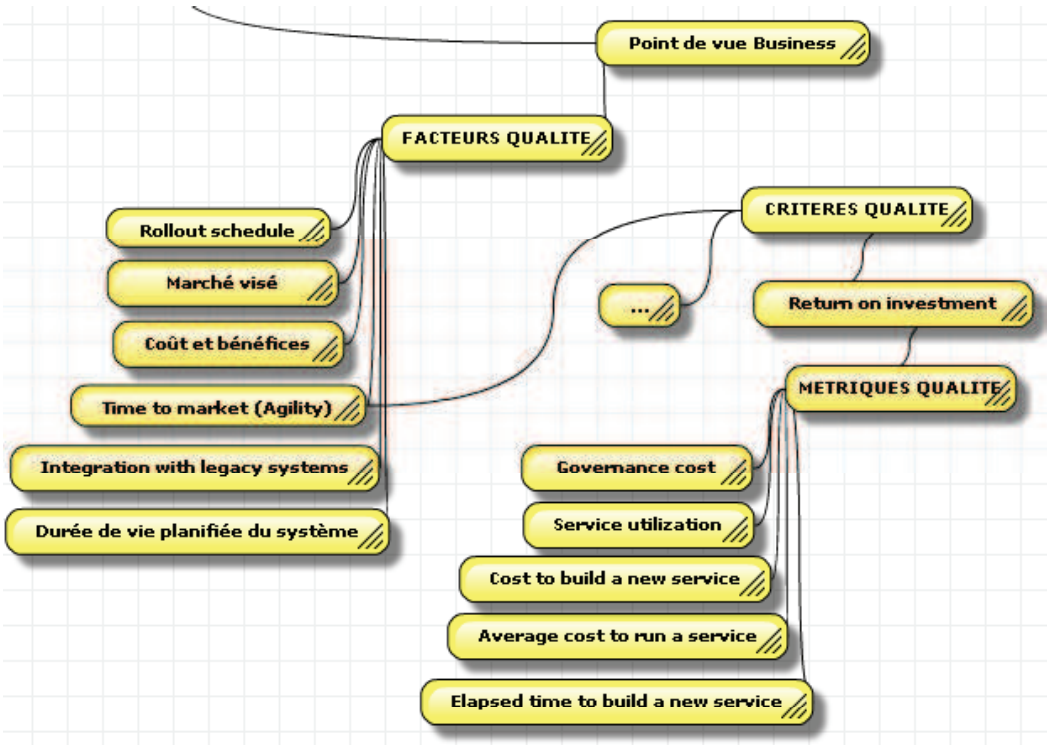

Fig. 3-2 – Début de déclinaison du point de vue business avec l'outil MindMap

Cette distinction à ce niveau de l'architecture permet d'isoler très précisément chacun des secteurs de l'architecture, qui sont les reflets des différents départements d'une organisation ordinaire. En effet les points de vue qualité, qui découlent directement des objectifs commerciaux de l'organisation, regroupent chacun une série d'attributs qualité appartenant à la même famille. Pour identifier les points de vue qualité de notre architecture, il suffit de constater quels sont les secteurs de l'organisation qui requièrent le plus d'importance pour une réussite optimale du projet (possibilités architecturales développées, intérêt accru du secteur business…).

[27] Le MindMap est un outil interne à BeOtic permettant de dessiner une carte heuristique (mindmap en anglais) à partir d'un mot ou d'un texte auxquels on associe des idées ou des concepts déclinant de l'élément central.

Ainsi, lorsque nous obtenons les résultats de l'évaluation et la valeur précise de la qualité pour chaque point de vue qualité, il nous est plus aisé de savoir quel secteur de l'architecture est affecté et comment directement intervenir pour rectifier le tir. Cela permet un important gain en termes de temps et de coûts ; en effet, puisque des intervenants sont affectés à chaque différent département au sein de l'organisation ; si nous obtenons une note de 90% pour la qualité du point de vue architectural mais de 10% seulement pour le point de vue business, nous alerterons en priorité les intervenants business et nous nous pencherons sur les attributs qualité dont ils sont en charge pour rectifier le tir.

3.2.2 Les facteurs qualité

Certains auteurs définissent le CBA (architecture à base de composants) avec les facteurs qualité : réutilisabilité et composabilité [CCL06]. En se basant sur cette analyse, et comme nous l'avions déjà présenté dans l'introduction ; nous définissons SOA avec les facteurs qualité **réutilisabilité**, **composabilité** et **dynamicité**. Ces trois attributs-qualité que nous avons définis comme étant les facteurs qualité clé pour SOA, représentent la quintessence qualitative qui a dirigé la définition des paradigmes objet, composant et service. La figure 3-3 expose cette analyse à travers un triptyque et offre une vision de haut niveau des points d'intérêt de SOA. Nous y voyons que la dynamicité hérite de la plus haute considération pour ce paradigme de développement logiciel, tandis que l'intérêt pour les deux autres facteurs qualité est moindre, mais toujours important.

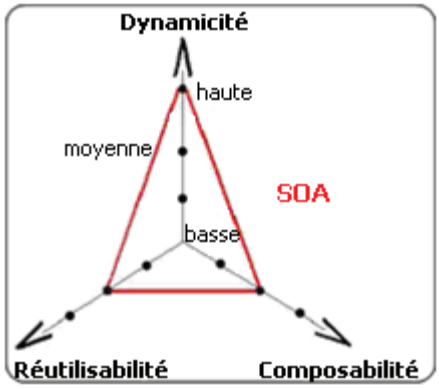

Fig. 3-3 - Les points d'intérêt de SOA

Nous le verrons plus loin, lorsque nous traiterons de la méthode SOAQE ; la seconde étape de la méthode SOAQE consiste à choisir un premier facteur qualité à étudier en profondeur et comme le montre la figure 3-3, la "dynamicité" est une problématique d'importance majeure lorsque nous travaillons sur les SOA, c'est précisément pourquoi nous avons choisi de nous concentrer particulièrement sur ce facteur qualité. Nous pouvons cependant tout de même signaler que les deux autres facteurs qualité, à savoir la "réutilisabilité" et la "composabilité" sont essentiels pour les trois paradigmes architecturaux considérés (objet, composant ou service).

3.2.3 Les critères qualité

Dans le cadre de notre travail, et après avoir identifié le facteur qualité déterminant pour SOA (la dynamicité), nous nous sommes intéressés au troisième niveau de décomposition de l'architecture et donc de notre modèle de qualité, à savoir, déterminer les critères définissant le facteur que l'on a choisi d'étudier. Ainsi, chaque facteur est composé de plusieurs critères que nous avons étudiés dans le cadre de notre travail.

Ces critères qualité appartiennent à des familles d'attributs qualité différentes et quelques uns d'entre eux sont communs aux trois facteurs qualité choisis pour nos travaux de recherché. Et puis, puisque nous nous sommes mis dans la peau d'un architecte, que l'on peut considérer comme un intervenant technique, nous avons délibérément choisi de concentrer notre travail sur les critères qualité techniques de

chacun des facteurs qualité étudiés. Dans cette optique, nous avons identifié six critères communs à chacun de nos trois facteurs qualité mais d'importance inégale en fonction du facteur considéré (voir figure 3-4).
Ces critères techniques rassemblent des éléments ayant des retombées majeures sur la qualité globale, à partir du processus de développement jusqu'au système produit.

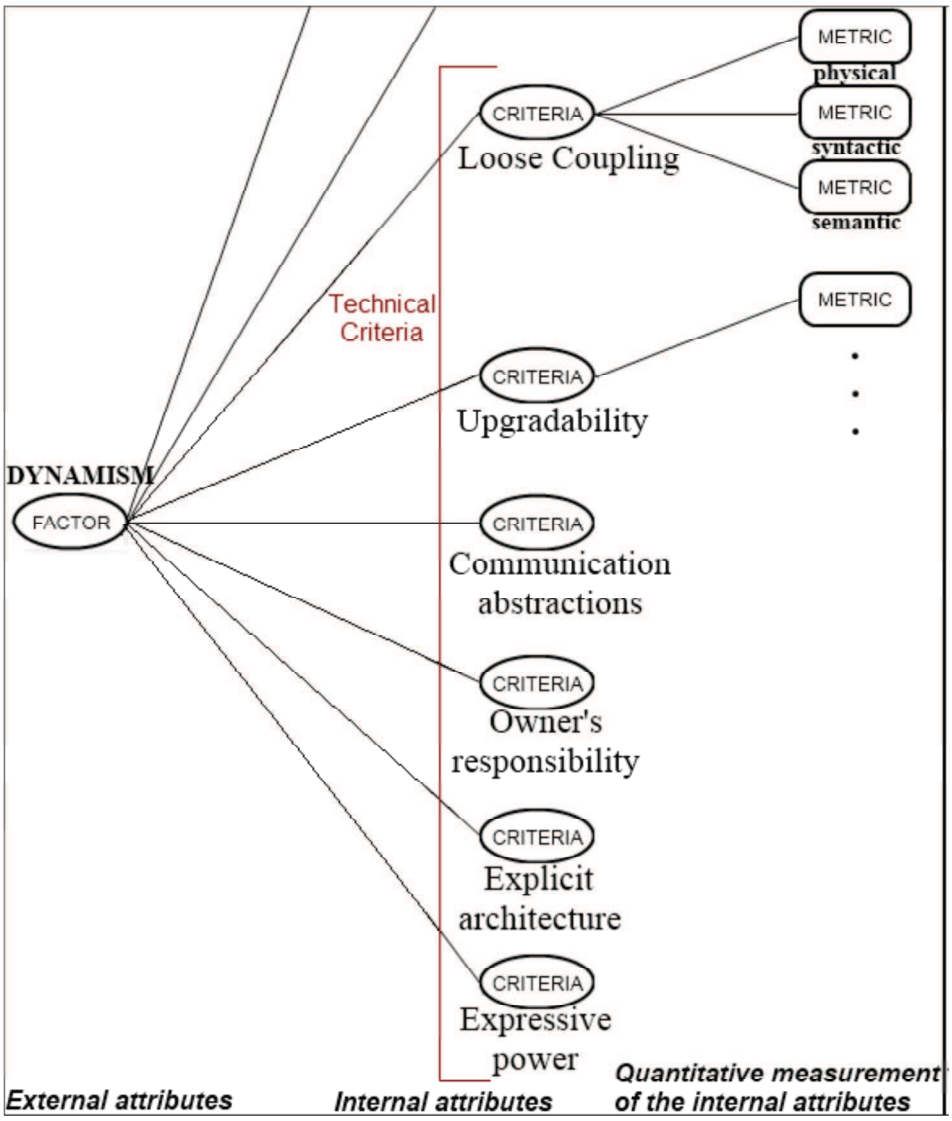

Fig. 3-4 - Le modèle de McCall appliqué au facteur qualité "dynamicité".

Couplage lâche : potentiel de réduction des dépendances entre produits et des dépendances entre processus.

Architecture explicite : capacité d'un paradigme à définir des vues architecturales claires d'une application, c'est-à-dire fournir les moyens d'identifier et d'expliciter les fonctionnalités associées aux produits qui composent l'application ainsi que les processus entre ces produits.

Abstraction de communications : capacité d'un paradigme à abstraire les communications des autres fonctions de l'application puis à appréhender et comprendre ces communications d'un seul tenant afin de pouvoir facilement les manipuler. Cette séparation des préoccupations est accrue par un découplage entre l'identification des communications par le design de l'architecture et la réalisation physique de ces communications. Cela représente la capacité de s'abstraire des problématiques physiques d'hétérogénéités des plateformes, du déploiement et de l'exécution, etc.

Pouvoir expressif : potentiel d'expression du paradigme en termes de capacité et d'optionalités de création. Il se base sur le nombre de concepts et de processus fournis pour spécifier, développer, manipuler, exécuter et implémenter des applications. Cette catégorie n'établit pas de jugement de valeur entre les différents concepts mais répertorie uniquement leur variété.

Pouvoir évolutif : potentiel d'un paradigme à faire évoluer ses produits et processus. Il se base sur une analyse et un jugement de valeur portés sur les différents processus qui supportent ces évolutions et sur leurs cibles.

Responsabilité propriétaire : correspond à la répartition des responsabilités entre fournisseurs et consommateurs. Ces responsabilités se focalisent sur l'entité logicielle réutilisée en terme de : développement, qualité de service, maintenance, déploiement, exécution, gestion. Cette répartition exprime le degré de liberté accordé aux consommateurs par le fournisseur.

Nous attribuons, comme nous l'avons indiqué plus tôt, différents degrés de considération à chacun de ces critères qualité en fonction du facteur qualité en question. Nos récents travaux de recherche [BOV12a] ont permis d'organiser

hiérarchiquement (sous trois niveaux distincts) ces critères qualité pour chacun des trois facteurs qualité (dynamicité, réutilisabilité et composabilité). Par conséquent, nous obtenons le triptyque du schéma 3-5 lorsque nous considérons l'ensemble des trois paradigmes.

Lorsque nous nous sommes concentrés sur la dynamicité, identifiée plus tôt comme étant le facteur qualité principal pour SOA, nos précédents travaux BOV12b nous ont permis de constater que le critère "couplage lâche" est le critère qualité le plus important pour ce facteur (voir le schéma 3-5), c'est pourquoi nous avons choisi de nous concentrer en détails sur celui-ci tandis que le critère "pouvoir expressif" est considéré comme le moins important.

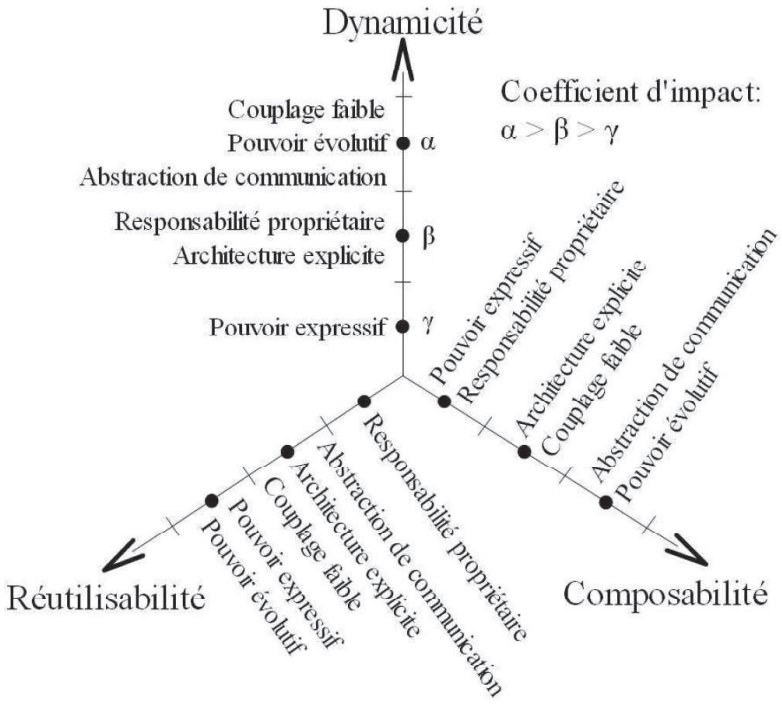

Fig. 3-5 - Expression des perspectives de réutilisabilité, de composabilité et de dynamicité.

3.2.4 Les métriques qualité

Les métriques qualité recouvrent des notions directement quantifiables et très précises que nous pouvons obtenir directement depuis le code source (nombre de lignes de code, nombre de fonctions etc.) avec des outils assez connus comme ceux que l'on a recensés dans la section 2.

Le couplage lâche est relativement bien connu par la communauté ; c'est ainsi que nous avons décidé au laboratoire de nous pencher sur ce sujet précisément. Nous avons alors pu isoler trois métriques qualité pour ce critère devant être considérés pour la quatrième et ultime étape de la décomposition du modèle de qualité, celle de la quantification et de l'évaluation des métriques qualité (le couplage sémantique {fort, faible, non prédominant} : *se base sur la description haut niveau d'un service qui est défini par l'architecte*, le couplage syntaxique {fort, faible}: *mesure les dépendances en terme de réalisation entre les services*, le couplage physique {α, β, γ et Δ avec $0 \leq \gamma \leq \beta \leq \alpha \leq \Delta$} : *mesure les dépendances entre les services réellement utilisés, en collaboration et en exécution.*

Le couplage physique réutilise les recherches existantes et se base sur des mesures empiriques [PRF07] telles que les appels de méthodes, les nombres de messages échangés, la complexité des messages, le nombre de services liés, etc.

Couplage sémantique

Couplage fort : Le service participe à une fonctionnalité essentielle du composite
Couplage faible : Le service participe à une fonctionnalité non essentielle du composite. La qualité globale n'est plus garantie si une ou plusieurs de ces fonctions sont retirées. Nous posons que si toutes ces fonctions non essentielles disparaissent le composite devient inutilisable.
Couplage non prédominant : un service abstrait et un composite sont en couplage non prédominant si ce service participe à une fonction non essentielle du composite et si le retrait de cette fonction n'a aucun impact significatif sur la qualité globale.

Couplage syntaxique

Couplage fort : un service abstrait est en couplage syntaxique fort avec sa solution concrète si cette solution (un service concret ou une composition de services concrets) représente l'unique possibilité de réalisation.
Couplage faible : un service abstrait et un service concret sont en couplage faible s'il existe plusieurs alternatives concrètes à la réalisation de ce service abstrait.

Couplage physique

Le couplage physique se focalise sur l'implémentation concrète du service. Cette implémentation correspond à une instance particulière du service où un choix a été fait sur les services concrets à utiliser. Une solution unique a été choisie pour remplir chacun des besoins exprimés par les services abstraits.
Nous pouvons ainsi dessiner un triptyque présentant clairement les métriques extraites et les niveaux d'acceptation pour chacune d'elles (voir le schéma 3-6).

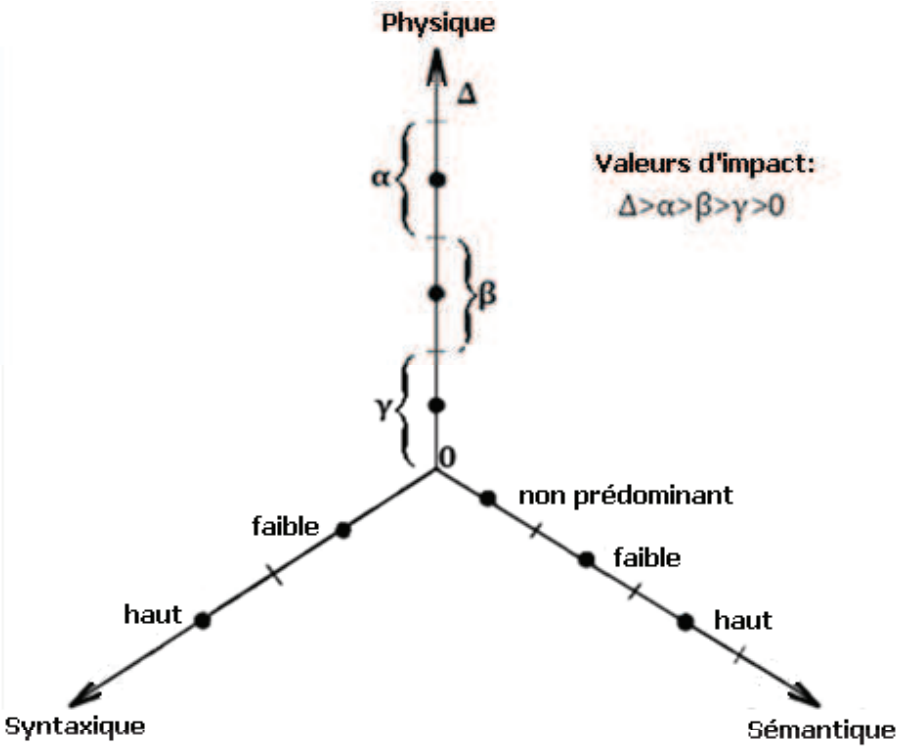

Fig. 3-6 - Métriques liées au couplage lâche.

3.2.5 Coefficients

L'identification des coefficients de pondération tient une place très importante dans notre expérience SOAQE dans la mesure où, avec la définition initiale des points de vue qualité de l'architecture par les décisionnaires humains en fonction des besoins de l'organisation, c'est la seule opération qui n'est pas automatisée et où l'intervention des intervenants humains (souvent techniques) est obligatoire. Pour statuer des coefficients nous devons systématiquement faire appel à l'homme car ceux-ci dépendent tout comme les points de vue qualité initiaux, entièrement des besoins de l'organisation.

En ce qui concerne la première étape de décomposition du modèle SOAQE, nous avons pour le moment conclu qu'il n'existerait pas de coefficients de pondération assignés aux points de vue qualité. Nous avons établi que par exemple, l'aspect business est tout aussi important que l'aspect architectural au seins d'une organisation, et que corrélativement, le coefficient de pondération du point de vue qualité business sera équivalent à celui du point de vue architectural. Pour le second niveau de décomposition de notre modèle, nos travaux nous ont conduit à conclure que pour SOA et pour les trois facteurs qualité sur lesquels nous avons travaillés, nous attribuerions, d'après le classement hiérarchique que nous avons dressé dans la section 2, un coefficient de pondération de "3" pour la "dynamicité" tandis que nous affecterions la valeur "2" pour la "réutilisabilité" et la "composabilité". Quant au troisième niveau de décomposition, les critères qualité, comme nous l'avons montré et justifié dans la section 2 nos travaux nous ont mené à lister les six critères qualité techniques choisis sous trois niveaux distincts d'acceptation, , et auxquels nous assignons respectivement les valeurs "3, 2 et 1", par conséquent, le "couplage lâche", "le pouvoir évolutif" et "l'abstraction de communication" seront affectés de la valeur "3". Le coefficient "2" sera attribué aux critères "responsabilité propriétaire" et "architecture explicite" tandis qu'on affectera la valeur "1" au critère qualité "pouvoir expressif".

Et finalement, les trois métriques étudiées peuvent être toutes affectées de la valeur "1" signifiant qu'elles sont toutes d'importances égales pour calculer le couplage lâche

d'une architecture orientée service. Ces coefficients de pondération ont été employés comme base pour notre expérimentation, notre méthode SOAQE et l'implémentation de notre outil. Ils ont été affectés aux attributs qualité en veillant à garder inchangés les rapports de proportionnalité entre les attributs qualité, puis validés par la communauté de l'ingénierie logicielle à travers multiples publications internationales. Nous pouvons sélectionner d'autres coefficients de pondération à condition que nous maintenions le même rapport de proportionnalité entre les attributs qualité considérés.

3.3 La méthode SOAQE

La méthode SOAQE est entièrement basée sur le modèle prévisionnel de qualité que nous avons présenté dans la section précédente. Chacun des niveaux de décomposition de l'architecture orientée service correspond à une étape de la méthode SOAQE à appliquer pour évaluer la qualité de l'architecture. Il en existe alors quatre aussi, en revanche, l'ordre des opérations est inversé à celui du modèle où nous identifions d'abord quels sont les points de vue qualité de l'architecture considérée. En effet, en ce qui concerne la méthode SOAQE, nous commençons d'abord par la fin de l'entonnoir, à savoir, mesurer la valeur des métriques qualité de l'architecture pour remonter jusqu'à l'ensemble des points de vue qualité

3.3.1 Les étapes relatives aux métriques qualité et aux critères qualité

Nous avons décidé de séparer cette section du manuscrit en deux paragraphes principaux ; l'un dédié aux métriques et aux critères qualité où est établi le plus important du travail quantitatif d'évaluation de la qualité et un autre moins conséquent consacré aux étapes de calculs des facteurs et des points de vue qualité.

La première étape de la méthode SOAQE nous incite à nous intéresser en premier lieu aux métriques qualité de notre architecture.

Comme nous l'avions indiqué dans la section 2, nous nous sommes durant nos travaux de thèse penchés sur les possibilités de quantifier le critère qualité couplage lâche et ses métriques qualité le constituant. En prenant comme point de départ une formule existante du domaine de "l'analyse préliminaire des risques" (voir la formule 3,1) [Mor02.], nos précédents travaux ont menés à l'identification d'une formule mathématique (voir formule 3,2) combinant les trois couplages étudiés : sémantique, syntaxique et physique.

Nota Bene : La formule simplifiée (voir la formule 3,1) habituellement utilisée dans l'industrie automobile, permet de mesurer le risque de défaut d'un composant d'une voiture. A est la criticité du composant de la voiture, B est la probabilité d'occurrence d'une défaillance sur ce composant et C est la probabilité de non détection de cette défaillance.

$$R = A * B * C \tag{3.1}$$

$$Coupling = \left\{\{(a),(b),(c)\}\right\} * \left\{\left(\left(\prod_{k=1}^{A}\prod_{i=1}^{N}\sum_{j=1}^{i}((P_s)_{kij})^{\alpha_{kij}}\right) * C_{phys}\right\} * \left\{P_{ndetect}\right\} \tag{3.2}$$

Nous avons associé ce concept de risque à notre vision du couplage. Corrélativement, la quintessence du couplage est l'expression des dépendances qui peuvent exister entre deux éléments et le principe de la dépendance définit qu'un élément ne peut pas être employé sans l'autre. Réduire le risque que le rôle défini par un service ne puisse plus être assuré permet de diminuer la dépendance de l'application par rapport à ce service et réduit de ce fait son couplage. Le calcul de ce risque prend en considération toutes les caractéristiques influençant le couplage en redéfinissant les trois variables A, B et C selon les couplages sémantique, syntaxique et physique. Le couplage global correspond à l'agrégation des trois couplages. Plus ce résultat est petit, plus le couplage est faible et plus le couplage est faible, plus l'architecture est claire et structurée.

Nota Bene : La criticité A ∈ [(a), (b), (c)] est affiliée au couplage sémantique. "a" si le service est seulement associé aux couplages non prédominants, "b" pour les

couplages non prédominants et les couplages faibles et "c" pour les couplages faibles, forts et non prédominants, alors que "Ps" est la probabilité de défaillance d'un service, "C_{phys}" est la valeur du couplage physique et "$P_{ndetect}$" est la probabilité de non détection d'une défaillance.

En appliquant aux attributs qualité connus les coefficients définis dans la section 3.2, nous obtenons l'arbre de la figure 3-7. L'état actuel de nos travaux de recherche nous permet de travailler en profondeur sur les chemins entourés par un cercle bleu sur le schéma 4-7 (le critère du couplage lâche).

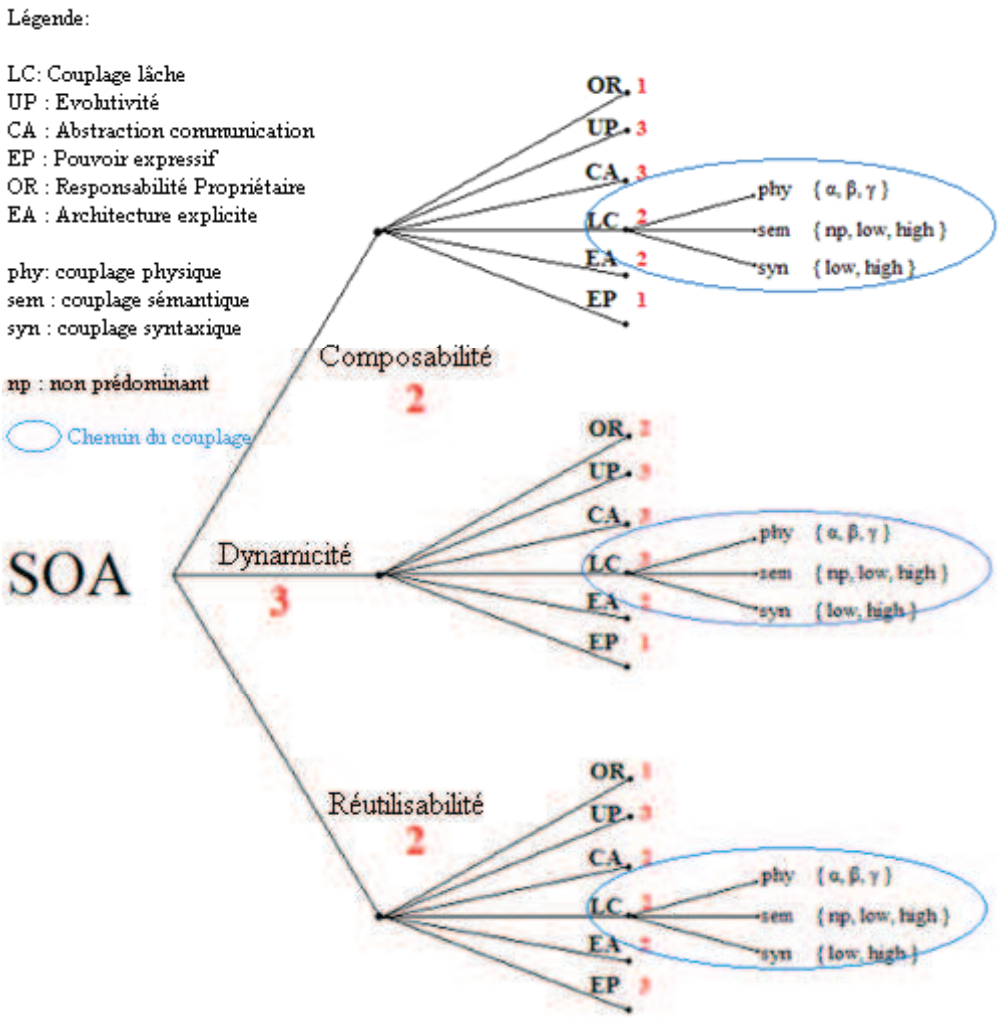

Fig. 3-7 - Arbre des attributs qualité pondérés à l'aide de coefficients.

Ainsi, la formule 3.3 ci-après permet de calculer la valeur de la qualité du point de vue architectural.

$$SOA = 3Dynamism * 2Reusability * 2Composability$$

$$SOA = 3(2OR + 3UP + 3CA + 3(1 - LC) + 2EA + EP) * 2(OR + 3UP + 2CA + 2(1 - LC) + 2EA + 3EP) * 2(OR + 3UP + 3CA + 2(1 - LC) + 2EA\ EP)$$

$$\begin{aligned} SOA = {} & 3(2OR + 3UP + 3CA + 3\ [1-(\ \{(a),(b),(c)\} * \left(\left(\prod_{k=1}^{\lambda} \prod_{i=1}^{N} \sum_{j=1}^{i} ((P_s)_{kij})^{\alpha_{kij}} \right) * C_{phys} \right) * P_{ndetect}\ )\ ] + 2EA + EP) * \\ & 2(OR + 3UP + 2CA + 2\ [1-(\ \{(a),(b),(c)\} * \left(\left(\prod_{k=1}^{\lambda} \prod_{i=1}^{N} \sum_{i=1}^{i} ((P_s)_{kij})^{\alpha_{kij}} \right) * C_{phys} \right) * P_{ndetect}\ )\ ] + 2EA + 3EP) * \\ & 2(OR + 3UP + 3CA + 2\ [1-(\ \{(a),(b),(c)\} * \left(\left(\prod_{k=1}^{\lambda} \prod_{i=1}^{N} \sum_{j=1}^{i} ((P_s)_{kij})^{\alpha_{kij}} \right) * C_{phys} \right) * P_{ndetect}\ )\ ] + 2EA + EP) \end{aligned} \quad (3.3)$$

NB: Plus le résultat du couplage lâche est bas, plus le couplage est faible. Réciproquement, plus le résultat de la qualité du point de vue architectural est élevé, plus la qualité est bonne. Le résultat de chaque critère qualité est exprimé en pourcentage, c'est pourquoi nous soustrayons à 1 le résultat du couplage lâche obtenu.

Cette formule générique du couplage est intéressante et peut directement être employée pour mesurer la qualité d'une architecture, le souci que nous avions avec cette formule réside dans la nature non automatisable de cette approche. En effet, pour pouvoir identifier chacune des métriques-qualité du couplage lâche, il faut analyser « à la main » chaque service constituant notre architecture et cette approche ne nous satisfaisait pas car nous souhaitions absolument restreindre au maximum l'intervention humaine dans le processus d'évaluation. Nous nous sommes alors orientés vers les outils existants que nous avions répertoriés dans la section 3 permettant la lecture automatique du code source de l'architecture soumise à l'évaluation afin d'en extraire une série de métriques automatiquement calculées.

Il existe des dizaines, voire des centaines de métriques dans notre domaine mais il n'en existe que quelques rares qui concernent directement ou indirectement le couplage lâche d'une architecture. L'idée était absolument de garder la décomposition de notre critère qualité couplage lâche en trois métriques (sémantique, syntaxique et physique) car la complémentarité de leur combinaison résume totalement l'expression du couplage d'une architecture. C'est ainsi que nos travaux

de recherche nous ont permis de répertorier les métriques existantes connues dans la communauté de l'ingénierie logicielle, représentant une expression de nos trois couplages (sémantique, syntaxique et physique) d'une architecture orientée service.

Les métriques existantes qui nous ont intéressées pour nos travaux proviennent majoritairement de trois suites de métriques très connues dans la communauté (La suite de Chidamber & Kemerer, la suite de McCabe et la suite de Henry & Kafura) auxquelles nous avons ajouté trois autres métriques indépendantes et sont :

- RFC
- CBO
- LCOM

} Chidamber & Kemerer Metrics Suite [CK91, CK94, CDK98]

- MPC
- Fan In

} Henry & Kafura Information Flow Metric [Hen96]

- Ce
- Ca
- V(G)

} McCabe Cyclomatic Complexity Metric [KKL+01]

RFC (Response For Class)

La métrique RFC correspond à la complexité générique de la hiérarchie des méthodes qui font une classe. Pour la calculer on compte le nombre de méthodes dans une classe et les méthodes qu'ils appellent directement. Plus le RFC est grand plus il va falloir faire de tests. La métrique RFC correspond au nombre de toutes les méthodes qui peuvent être appelées en réponse à un message à un objet de la classe ou par une certaine méthode dans la classe. Ceci inclut toutes les méthodes accessibles dans la hiérarchie de la classe. Cette métrique RFC compte les occurrences des appels à d'autres classes à partir d'une classe particulière [CK94].

CBO - Coupling Between objects

La métrique CBO mesure à quel point une classe dépend d'autres classes. Elle correspond au nombre de classes externes dont les membres sont appelés, lus ou utilisés comme types par des membres de la classe courante. Un résultat de la métrique CBO limite la disponibilité de la classe pour qu'elle soit réutilisée et engendre également des efforts supplémentaires en ce qui concerne les phases de test et de maintenance. Le nombre total de classes qu'une classe référencie plus le nombre de classes qui référencient la classe sans compter les doublons. Il a existé de nombreuses définitions pour la métrique CBO ces dernières années et la plus courante stipulait que CBO correspondait au nombre de classes qu'une classe référencie [CK91]

LCOM – Lack of Cohesion in Methods

Cette métrique mesure à quel degré les variables membres sont utilisées pour partager des données entre les fonctions membres. Elle compte les paires de méthodes de classe qui n'accèdent à aucune des mêmes variables de classe auxquelles on soustrait le nombre de paires qui elles le font. Plus la valeur de la métrique est élevée, plus la cohésion de l'architecture est faible. Cela est corrélé avec une encapsulation plus faible, et est un indicateur que la classe est candidate à une décomposition en sous-classes. LOCM est une mesure retranscrivant comment les méthodes d'une classe interagissent avec les données dans une classe [CDK98].

Message Passing Coupling (MPC)

Cette métrique mesure le nombre de messages passes entre les objets d'une classe. Un résultat élevé indique que le couplage est élevé entre cette classe et les autres classes du système. Ceci rend les classes plus dépendantes les unes des autres ce qui augmente la complexité générale du système et rend la classe plus difficile à modifier [Hen96].

Fan In

Cette métrique correspond au nombre de classes depuis lesquelles une classe est dérivée. Autrement dit c'est le nombre de classes qui référencent une classe [Hen96].

Ce – Efferent Coupling

Cette métrique inclue tous les types dans la source du package mesuré faisant référence aux types qui ne sont pas dans le dit package. Un résultat élevé de la métrique peut indiquer que le package est instable puisqu'il dépend de la stabilité de tous les types desquels il est couplé. Les travaux existant dans le domaine recommandent une limite maximale de 20. Ce résultat peut être baissé en extrayant des classes depuis la classe originelle, pour qu'ainsi elle puisse être décomposée en plus petites classes [Gor06].

Ca – Afferent Coupling

Cette métrique détermine le nombre de classes et d'interfaces depuis d'autres packages qui dépendent de classes provenant du package analyse [Gor06].

V(G) – Cyclomatic complexity

Cette métrique est une mesure de la complexité de la structure de décision d'un module. C'est le nombre de chemins linéairement indépendants et donc, le nombre minimal de chemins qui doivent être testés. Cette métrique est souvent référencée comme étant simplement la complexité du programme [KKL+01].

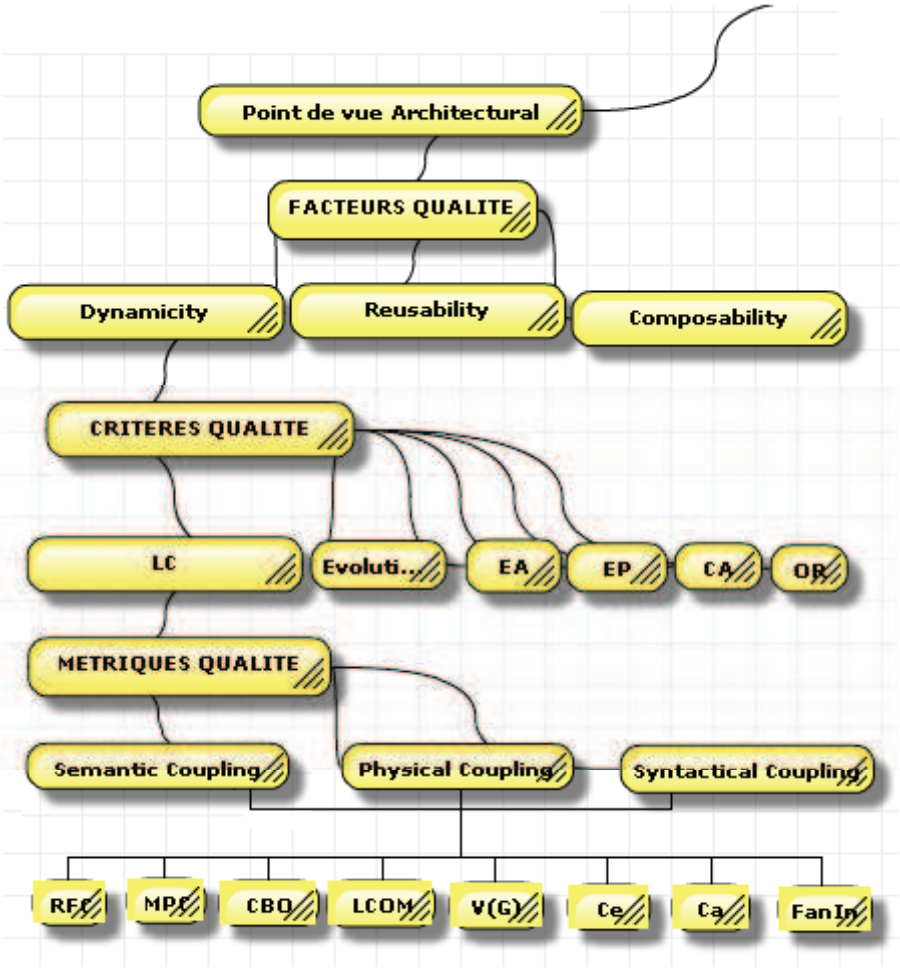

Fig. 3-8 - Déclinaison du point de vue architectural avec l'outil "MindMap"

Les métriques que nous avons choisi sont celles qui, d'après nos études bibliographiques, relatent le mieux le concept du couplage lâche dans une architecture orientée service (voir figure 3-8) mais comme nous le verrons dans la section 4 dédiée à la réalisation de notre prototype, nous avons conçu un outil totalement modulable au point de laisser la liberté à son utilisateur de modifier l'arborescence de l'architecture depuis les points de vue qualité jusqu'aux métriques.

Afin d'obtenir un pourcentage de la qualité de la métrique, nous nous aidons des seuils en étudiant les travaux établis sur les métriques dans la littérature. Ainsi, par exemple, lorsque nous nous penchons sur la métrique liée à la complexité

cyclomatique d'une SOA (V(G)), nous savons que plus elle est basse mieux c'est et que son seuil maximal d'acceptation est évalué à 30 (valeur sans unité).

Ainsi, pour obtenir un pourcentage précis de la complexité cyclomatique de l'architecture évaluée ; nous considérons qu'une valeur résultat de 0 pour V(G) correspond à la valeur optimale, soit 100% ; à contrario, une valeur de 30 pour V(G) correspond à une valeur de 0% pour notre outil. Ainsi graduellement entre les valeurs de 0 et 30 nous obtenons proportionnellement 86,6% si l'outil nous renvoie 4 pour V(G), 50% s'il nous renvoie 15 et 10% s'il nous renvoie 27.

Puisqu'il existe des seuils pour toutes les métriques existantes afin qu'on puisse évaluer les résultats de l'évaluation, nous procédons de la même façon avec les autres métriques que nous considérons.

Nous établissions dans la section 3.2.4 que toutes les métriques liées au couplage lâche étaient d'égale importance pour une SOA, ce qui implique que nous pondérons toutes nos métriques d'une valeur neutre « 1 » et ainsi, pour déterminer la valeur en pourcentage de la qualité du critère qualité couplage lâche, il suffit de faire une moyenne des pourcentages que l'on obtient pour chacune des métriques qualité considérées.

Si nous considérons, par exemple, que nous obtenons les valeurs des métriques suivantes :

V(G) = 27%

Ca = 42%

Ce = 39%

LCOM = 21%

RFC = 75%

CBO = 60%

MPC = 81%

Fan in = 10%

$$\frac{27+42+39+21+75+60+81+10}{\text{Somme des métriques considérées}} = \mathbf{44,4\%}$$

Le couplage lâche de notre architecture obtient une valeur pour sa qualité évaluée à 44.4%

3.3.2 Les étapes relatives aux facteurs qualité et aux points de vue qualité

Une fois que nous avons pu déterminer une valeur précise de chacune des métriques qualité que nous avons recensées dans notre architecture, le plus gros du travail a été réalisé. En effet comme nous l'avons vu dans le chapitre précédent, pour déterminer la valeur de l'ensemble des critères qualité de notre architecture, il suffit simplement d'établir la moyenne des métriques qualité existantes sous chacun des critères puisque aucun coefficient n'a été attribué aux métriques qualité dans la mesure où il a été conclu d'après nos recherches bibliographiques qu'ils sont tous d'importance égale, et ce, quel que soit le critère qualité considéré. Et bien le travail restant est tout à fait semblable, exception faite que cette fois les attributs qualités sont pondérés. Effectivement, si nous étudions une nouvelle fois l'arbre de la figure 3-7 , nous constatons que pour déterminer la valeur du facteur qualité Dynamicité, il est nécessaire d'établir l'agrégation des critères qualité que l'on connaît tout en tenant bien sûr compte de leur coefficient de pondération.

Facteur Dynamicité = 2OR + 3UP + 3CA + 3LC + 2EA + 1EP

Considérons, par exemple que nous obtenons pour chacun des critères qualité que l'on connait les valeurs suivantes :

OR = 32%

UP = 56.3%

CA = 21.9%

LC = 44.4%

EA = 72.7%

EP = 58%

$$\frac{2 \times 32 + 3 \times 56{,}3 + 3 \times 21{,}9 + 3 \times 44{,}4 + 2 \times 72{,}7 + 58}{\text{Somme des critères qualité pondérés}} = \mathbf{49.9\%}$$

Dans le cadre de notre exemple, le facteur qualité dynamicité de notre architecture obtient une valeur de 49.9% pour sa qualité.

Pareillement, pour déterminer la valeur des points de vue qualité de notre architecture, nous procédons de la même manière. Si nous nous concentrons sur le point de vue architectural et que nous consultions également l'arbre du schéma 4-7 , nous constatons que :

Point de vue qualité Architectural = 3Dynamicité + 2Réutilisabilité + 2Composabilité

Si nous considérons à titre d'exemple les valeurs de la qualité des facteurs suivantes :

Dynamicité = 49.9%

Réutilisabilité = 32.1%

Composabilité = 38,6%

$$\frac{3 \times 49{,}9 + 2 \times 32{,}1 + 2 \times 38{,}6}{\text{Somme des facteurs qualité pondérés}} = \mathbf{41{,}6\%}$$

Nous obtiendrions pour la valeur du point de vue architectural la valeur de 41.6% comme valeur réelle de sa qualité et c'est ainsi qu'à partir des métriques qualité que l'on a identifiées jusqu'aux points de vue qualité de l'architecture, nous sommes en mesure à l'aide de la méthode SOAQE de déterminer la qualité générique de l'architecture. Aussi, il est important de stipuler que la méthode SOAQE doit être suivie pour chacun des points de vue qualité que l'on a identifiés après avoir analysé

les objectifs de l'organisation et l'ensemble des facteurs, critères et métriques composant les points de vue qualité.

3.4 Conclusion

Tout au long de cette section, nous avons présenté le modèle SOAQE et ses différents niveaux d'attributs qualité hiérarchisés et au vu des informations que nous avons présenté nous pouvons en conclure que le modèle SOAQE pourrait convenir comme la plupart des modèles de qualité que nous avons présenté dans la section 1.4.2 à la vue utilisateur, à la vue de fabrication et la vue du produit que nous avons détaillées dans la conclusion du chapitre 1 (section 1.5) car celui-ci se focalise sur les deux courants de la qualité que nous avons définis dans la section 1.4, c'est-à-dire non seulement sur la conformité de la spécification mais aussi sur des réponses aux besoins des utilisateurs ce qui permet une couverture assez complète de la notion de qualité logicielle.

En ce qui concerne la méthode SOAQE, nous avons pu découvrir son mode de fonctionnement à travers un cas d'exemple où nous avons pu déterminer une note en pourcentages pour le point de vue architectural. Bien que cet exemple d'utilisation de la méthode SOAQE (et les valeurs utilisées et obtenues) ne permette pas de la valider, il était utile pour tenter de comprendre son fonctionnement générique. Afin d'apporter un regard critique sur le travail que nous avons fourni, nous avons voulu opérer de la même manière qu'avec les méthodes d'évaluation que nous avons présentées dans la section 2.3.2 de ce manuscrit. En effet, nous avons soumis notre méthode aux huit critères de comparaison que nous avons listés dans le tableau 2.2 et avons obtenu les résultats du tableau 3-1 suivant :

Tab. 3-1 – Les critères de comparaison appliqués à la méthode SOAQE

Méthode	SOAQE
C1	Mesure quantitative de la qualité à l'aide de critères pondérés par l'Homme.
C2	Dynamicité, composabilité, réutilisabilité
C3	La liste d'intervenants techniques minimum que nous avons dressée dans la section 2.2.1 et dans le cas de notre utilisation au sein de BeOtic, nous avions en tant que producteur du système, un architecte logiciel et en tant que consommateurs de système, deux directeurs commerciaux, un développeur des utilisateurs de services et un développeur chargé de maintenance. Tous ces intervenants se sont également glissés dans la peau d'utilisateurs finaux lorsqu'il s'agissait d'interpréter les résultats de l'évaluation.
C4	Valeurs des métriques qualité calculées à partir des sources
C5	Oui
C6	Semi-automatisée, évaluation quantitative et non approximative, identification des zones de complexité potentielle, méthode ouverte à toute description architecturale,
C7	Se limite à ce moment aux aspects techniques de l'architecture.
C8	Notre plateforme de développement à BeOtic et tout système d'information en général

Au vu du tableau 3-1 précédent, notre méthode SOAQE répond convenablement à la majorité des critères de comparaison et par conséquent semble plutôt complète en ce qui concerne le traitement de l'aspect technique de l'architecture ; nous allons, à travers la section 4 suivante, justement détailler l'outil que nous avons élaboré et lui soumettre un cas d'étude extrait d'un projet existant à BeOtic afin de pouvoir valider le modèle, la méthode et le prototype SOAQE.

Chapitre 4

RÉALISATION ET EXPÉRIMENTATION

4.1 Introduction

Cette nouvelle section du manuscrit est entièrement dédiée à la phase d'implémentation dans les locaux de BeOtic.

Nous y présentons quatre sous-parties dans lesquelles nous expliquons d'abord dans la première les choix relatifs à l'aspect "*réalisationnel*" de notre outil.

La seconde sous-partie est quant à elle dédiée à la présentation des données de tests qui permettront d'illustrer notre outil. Ces données émanent d'un projet existant de la société BeOtic et incluent le contenu des sources d'un des clients de la société. La troisième sous-partie traite quant à elle de la première fonctionnalité de notre application : la visualisation de données tandis que la dernière concerne la seconde et principale fonctionnalité de notre outil, à savoir, l'évaluation de la qualité de l'architecture.

4.2 Réalisation du prototype SOAQE

4.2.1 L'outil SOAQE

Dans cette section du manuscrit, nous présentons l'outil SOAQE, un outil qui est entièrement basé sur le modèle et la méthode éponymes et qui a été établi en coopération avec la Société BeOtic qui à terme compte l'intégrer à un service proposé à ses clients.

4.2.2 Architecture technique

L'outil SOAQE est basé sur une application client-serveur. Pour l'implémenter, nous avons utilisé la plateforme de développement de BeOtic afin que l'on puisse l'y intégrer très simplement puisqu'à terme le but est de le commercialiser.

- Nous avons choisi de travailler avec un serveur ayant été implémenté à l'aide de la technologie **Java** essentiellement car les serveurs de BeOtic sont en Java ; l'intérêt est qu'à terme nous incrustions notre solution dans les serveurs de l'entreprise. Il communique avec une base de données MySQL[28] (nous avons choisi cette technologie pour son évolutivité, sa performance et son support de production) par l'intermédiaire de la technologie **DAO**[29] (pour Data Access Object) qui fournit une interface abstraite permettant l'exécution d'opérations liées aux données de l'application sans pour autant exposer les détails de la base de données MySQL [Ber05, Sun02]. En quelques sortes, DAO agit comme un intermédiaire entre l'application et la base de données. Cette technologie nous a fortement intéressés car elle peut être utilisée sur une large palette d'applications où la sauvegarde de données est requise [Rom02].

- Le coté client de l'application a été implémenté en utilisant la technologie **Flex** (une technologie qui tourne sur le lecteur Flash et qui est gérée par Adobe). Nous avons opté pour cette technologie car il est plus accommodant de développer des RIA (*Rich Internet Application*) avec Flex qu'avec Html et JavaScript par exemple. Aussi, les applications Flex peuvent facilement être exécutées hors ligne, sur des applications bureautiques par exemple [Sun12]. Le client Flex communique avec le serveur par l'intermédiaire de la technologie Adobe **BlazeDS**[30] qui est un logiciel open source facilitant les échanges entre Flex et Java en s'interposant entre les deux à la manière d'un proxy [CS08]. Il permet de localiser et d'invoquer des services JAVA, il supporte les appels RPC et les échanges de messages entre les deux plateformes [Tiw09].

---

[28] http://www.oracle.com/fr/products/mysql/index.html
[29] www.oracle.com/technetwork/java/dataaccessobject-138824.html
[30] http://opensource.adobe.com/wiki/display/blazeds/BlazeDS/

Le schéma 4-1 décrit l'architecture de l'outil SOAQE.

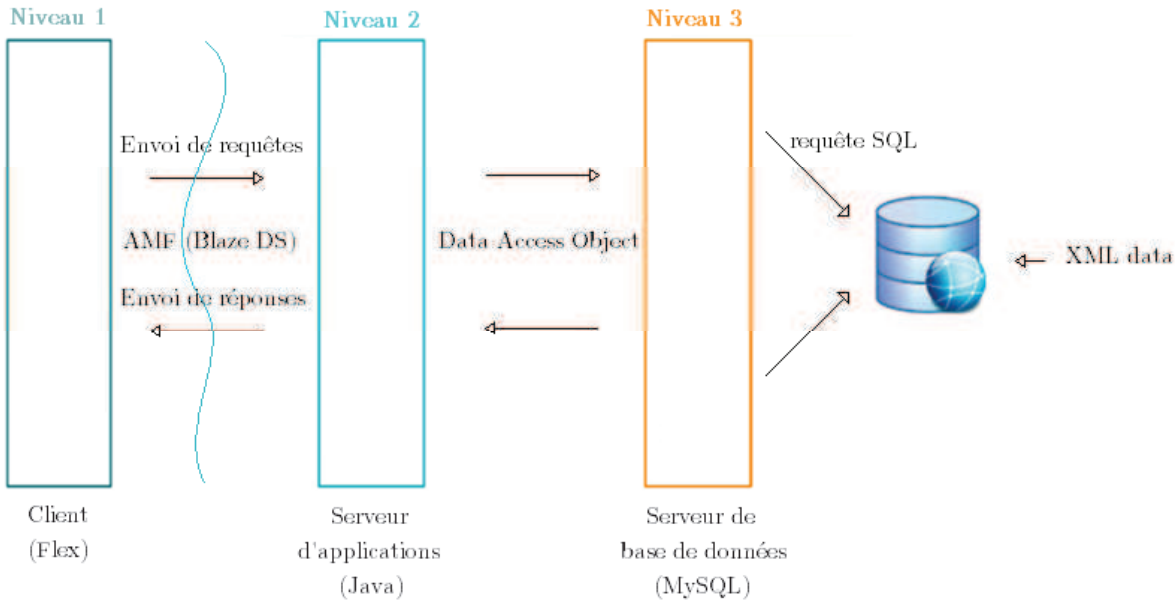

Fig. 4-1 - Architecture de l'outil SOAQE.

Une fois notre architecture définie et mise en place, la prochaine étape consistait à étudier les données que nous allions traiter puis sauvegarder dans la base de données.

4.3 Données utilisées

4.3.1 Introduction

Pour cette partie du manuscrit, nous allons revenir sur les données initiales du projet que l'on avait en notre possession pour valider notre application.

Ces données sont les sources du projet d'une très grande compagnie de plus de 100 000 employés (BeOtic ne préfère pas divulguer son identité) qui travaille avec BeOtic.

Nous allons revenir sur toutes les opérations servant de base à notre prototype jusqu'à l'obtention de la valeur des métriques de notre architecture.

Sommairement, nous partons des sources du projet, auxquelles on applique un outil qui nous retourne un ensemble de métriques accompagnées de leurs valeurs respectives. Plus concrètement, notre société BeOtic a implémenté son propre outil appelé BeoMetrics pour calculer des métriques du code (telles que V(G), LCOM, CCN…) à partir du code source du projet; cet outil fonctionne comme JMetric[31] et délivre le résultat de ses calculs sous formes de plusieurs sources XML et CSV qui contiennent les valeurs de plusieurs métriques pour chaque méthode, classe et package de l'architecture du projet de l'utilisateur. Il existe, comme nous l'avons présenté dans la section 2.4 beaucoup d'outils qui reproduisent le même travail mais aucun d'entre eux ne nous retournait les valeurs de l'ensemble des métriques que l'on voulait mesurer à BeOtic. L'intérêt majeur d'utiliser son propre outil (BeoMetrics) tient dans l'idée de pouvoir incorporer au fil de l'eau les métriques qui nous sont utiles exclusivement.

Dans la section suivante, nous analysons les résultats de BeoMetrics en expliquant très précisément comment les valeurs ont été obtenues.

Puis, en s'appuyant sur l'étude des seuils que l'on a effectuée pour les métriques à travers la littérature, nous déterminerons les valeurs de quelques unes des métriques (en pourcentages) nous concernant directement pour notre application et qui nous servent de socle pour toutes les opérations suivantes de l'outil SOAQE.

4.3.2 Application de BeoMetrics

Au moment de commencer l'implémentation de notre outil SOAQE, BeOtic a mis à ma disposition les sources d'un projet de grande envergure pour valider nos réalisations futures.

[31] http://sourceforge.net/projects/jmetric

Dans un même dossier nous avons regroupé ces sources (dossier data de la figure 4-2) et l'outil BeoMetrics (fichier.bat de la figure 4-2).

L'outil BeoMetrics étant déjà opérationnel, nous n'avions plus qu'à l'exécuter pour obtenir le calcul des différentes métriques (voir figure 4-2)

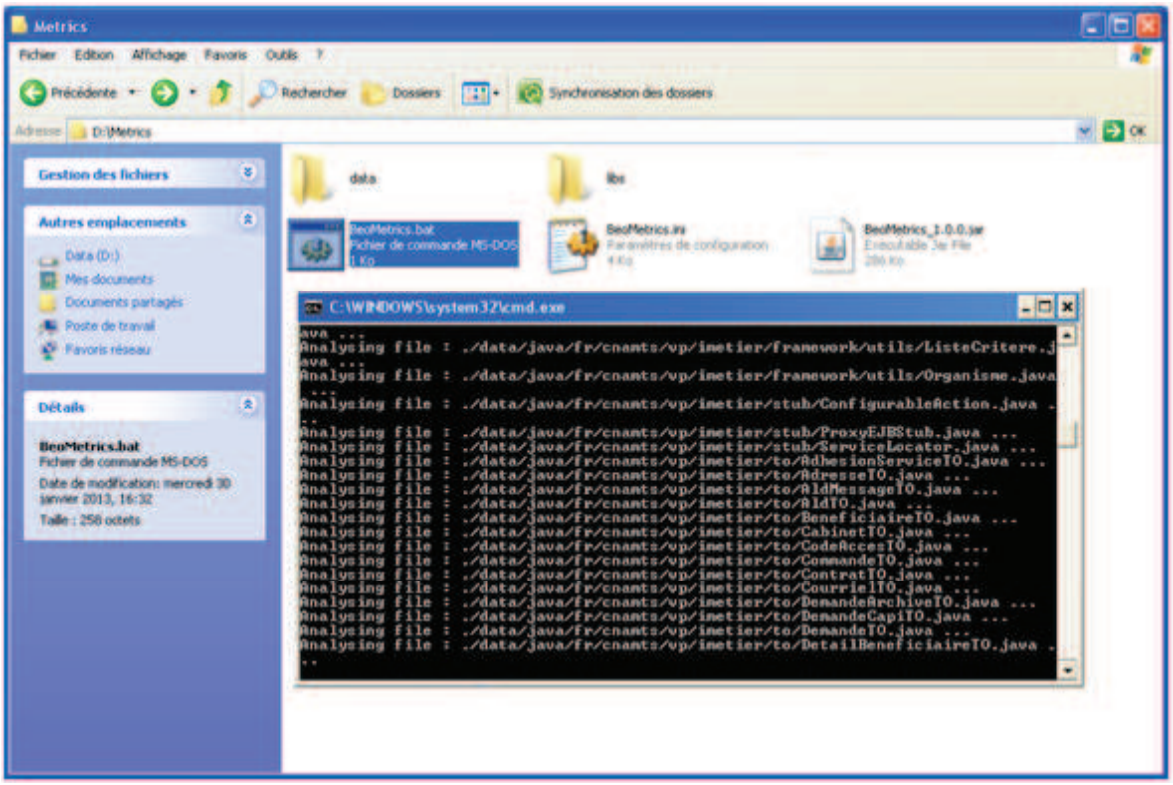

Fig. 4-2 - Analyse de BeoMetrics en cours

Une fois l'analyse de BeoMetrics terminée (nombre de classes, de méthodes du projet puis durée de l'analyse en guise de résultat), un nouveau dossier "result" regroupant toutes les valeurs des métriques apparaît (voir figure 4-3).

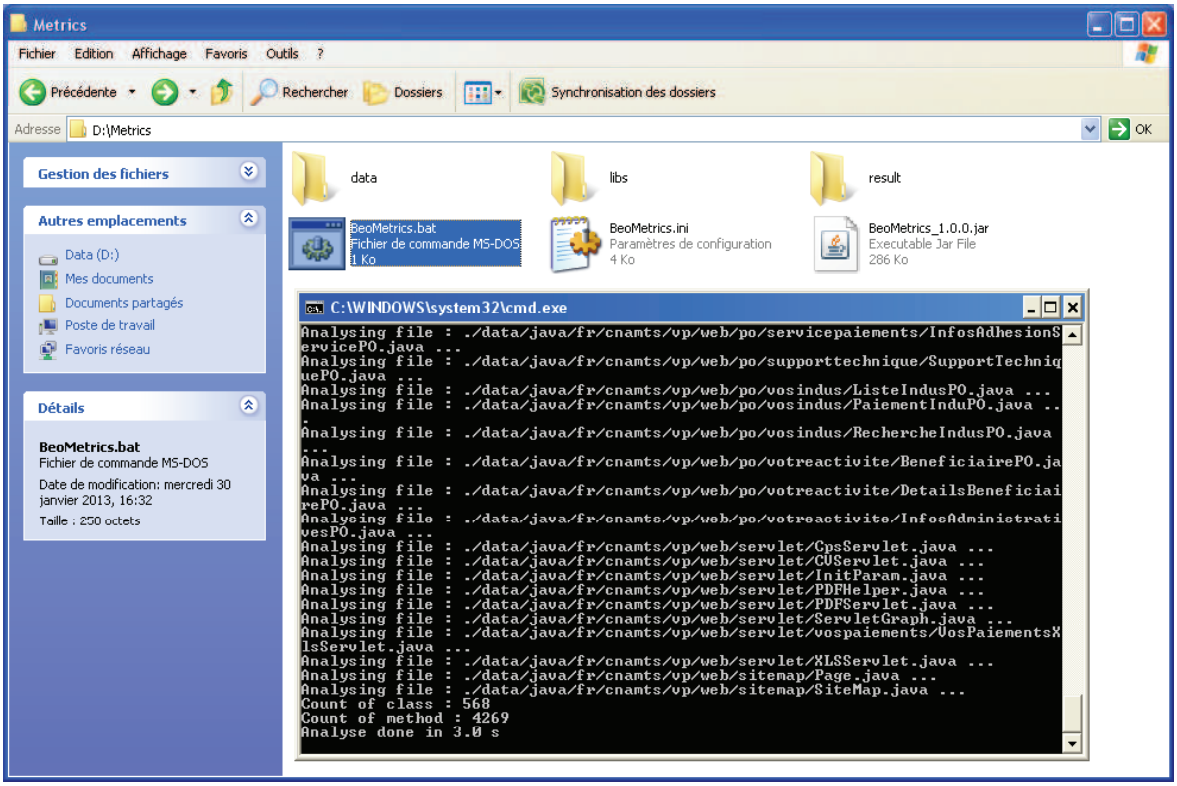

Fig. 4-3 - Analyse terminée

Comme le montre la figure 4-4, ce dossier de résultats contient plusieurs sources :

classes.xml : regroupe l'ensemble des valeurs des métriques calculées au niveau des classes.

classesDetails.xml : ce document relate l'ensemble des classes des sources et les méthodes qu'elles contiennent accompagnées de leur position (numéro de ligne) dans la classe.

methods.xml : regroupe l'ensemble des valeurs des métriques calculées au niveau des méthodes.

packages.xml : regroupe l'ensemble des valeurs des métriques calculées au niveau des packages.

BeoMetrics_Result.csv : regroupe l'ensemble des valeurs des métriques calculées, toutes sources confondues.

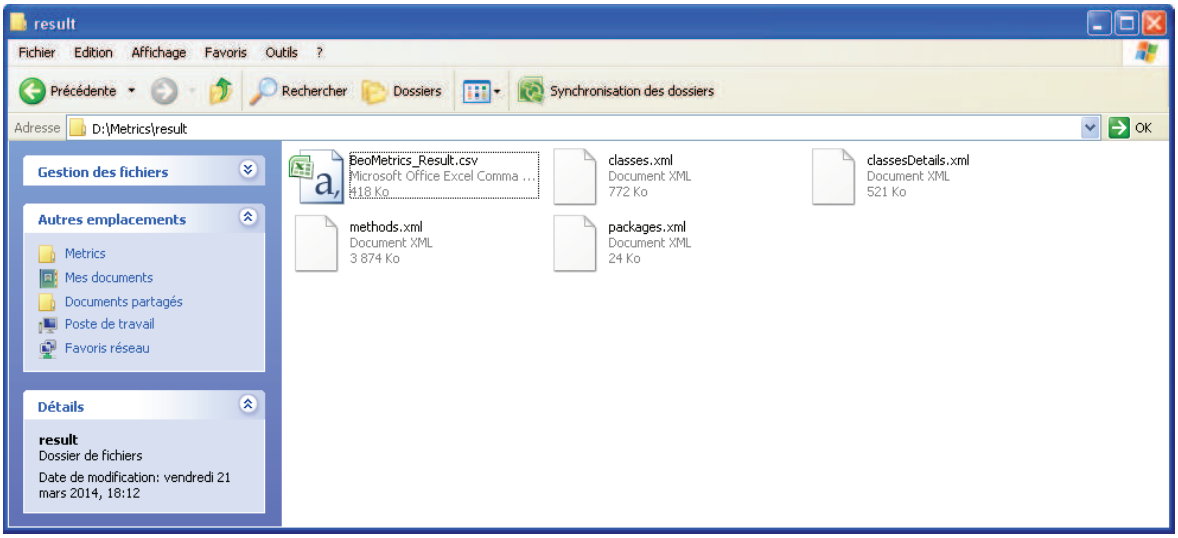

Fig. 4-4 - Dossier des résultats de BeoMetrics

4.3.3 Analyse des résultats

4.3.3.1 Introduction

Cet outil assez puissant nous fournit, en guise de résultats, des sources très complètes.

Ayant choisis de nous positionner au niveau des classes uniquement, les sources qui attiraient notre curiosité étaient évidemment les documents qui se rapportaient aux classes du projet mais certaines données n'étant pas pertinentes pour la suite de nos opérations, tels que les résultats contenus dans **classesDetails.xml** (décrit plus tôt), nous nous sommes concentrés sur le document **classes.xml** qui regroupe les valeurs de vingt-sept métriques pour chacune des 568 classes du projet.

4.3.3.2 Obtention des valeurs des métriques

Nous avons dans un premier temps tenu à vérifier la pertinence des données contenues dans le document **classes.xml.**

Pour ce, nous avons parcouru le document et analysé plusieurs données relatives aux classes du projet.

Nous avons observées les valeurs des métriques calculées et avons comparé les résultats avec ceux que l'on pouvait déduire en analysant le code source du projet.

Pour témoigner de l'exactitude des résultats, considérons la classe "AbstractFeuilleBlanchePS" pour laquelle BeoMetrics nous retourne les métriques de la figure 4-5 suivante :

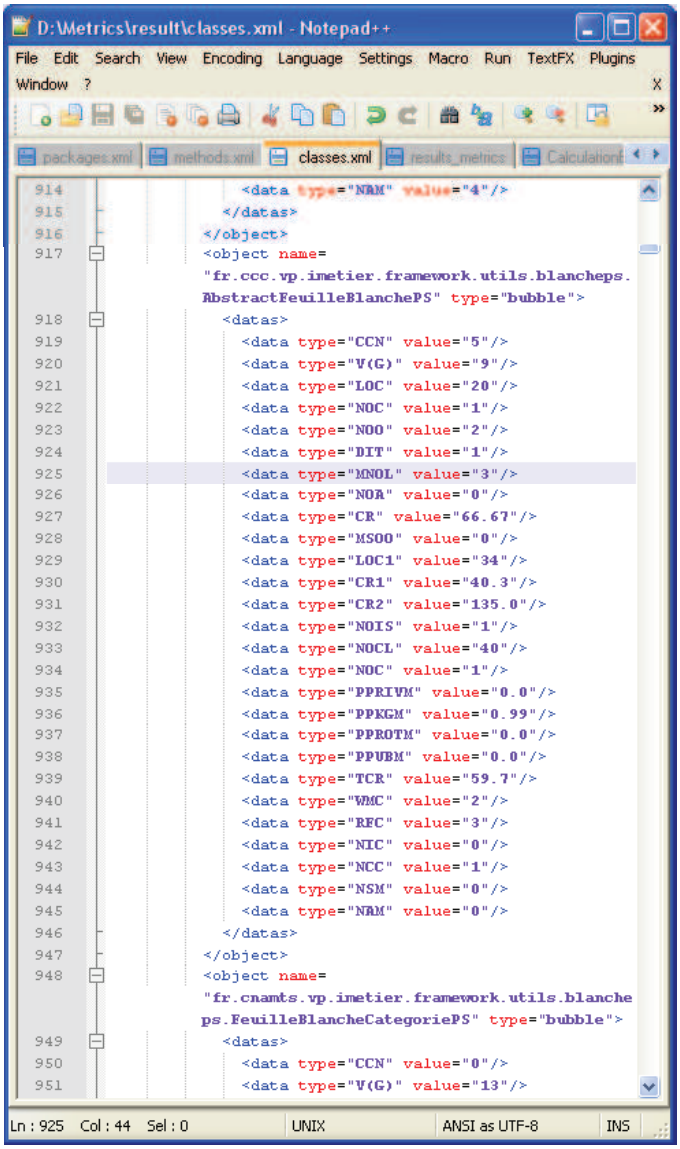

Fig. 4-5 - Valeurs des métriques de la classe "AbstractFeuilleBlanchePS"

Maintenant, concentrons-nous sur l'implémentation de la classe "AbstractFeuilleBlanchePS" que nous pouvons voir sur la figure 4-6 suivante :

Fig. 4-6 - Code de la classe "AbstractFeuilleBlanchePS"

Afin de certifier de la pertinence des résultats, nous pouvons comparer des données simples et rapidement analysables comme les lignes de code effectives, les lignes de code à l'intérieur de la classe et le ratio des commentaires de la classe (respectivement LOC, LOC1 et CR1 du document **classes.xml** de la figure 4-5).
Ce dernier nous affiche la valeur de "20" pour LOC ; or lorsque nous comptons les lignes effectives de code dans la source "AbstractFeuilleBlanchePS" (c.-à-d. hors lignes de commentaires et lignes blanches) nous en comptons effectivement tout juste 20. Nous obtenons la valeur "34" pour LOC1, or la définition de cette classe qui commence à la ligne 28 (non incluse) jusqu'à la ligne 63 compte bel et bien 34 lignes.

Et finalement nous obtenons la valeur de "40.3" pour la métrique CR1 : nous comptons comme nous pouvons le voir sur la figure 32, très exactement 28 lignes de commentaires pour une source de 68 lignes au total ; ce qui représente très exactement un ratio de 40.3% de commentaires.

4.3.4 Diagramme de classes

La véracité des résultats de BeoMetrics ayant été prouvée, la prochaine étape consistait à établir une base de données MySQL claire, complète et définitive dans laquelle nous allions logiquement stocker les données contenues dans ces sources pour faciliter la recherche et l'accès à ces données depuis l'application. Pour concevoir cette base de données, il fallait préalablement dessiner le diagramme de classes de notre application.

4.3.4.1 Tables

Ce diagramme de classes expose chacune des tables de notre base de données, leurs attributs et leurs interconnexions (voir fig. 4.2).

Il est composé des tables principales *PointofView*, *Factor*, *Criterion* et *Metric* correspondant à nos attributs qualité décomposant l'architecture orientée service. Ces tables contiennent les noms des attributs. Nous avons également conçu les tables *FactorValue*, *CriterionValue* et *MetricValue* contenant, respectivement, les valeurs

de tous les facteurs, critères et métriques de l'architecture : la table *MetricValue* contient les valeurs de chacune des métriques obtenues après application de l'outil BeoMetrics ; la table *CriterionValue* contient la valeur du critère en question après agrégation des métriques tandis que la table *FactorValue* contient la valeur du facteur concerné après agrégation et pondération des critères de ce facteur.

Les tables *PointofViewFactorLink* et *FactorCriterionLink*, contiennent, respectivement, les valeurs des coefficients des facteurs, puis des critères de l'architecture (nous avons défini que nous ne pondérions pas les métriques donc il n'existe pas de table *CriterionMetricLink*). Ces tables servent de "jonction" entre les différents niveaux des attributs qualité de l'architecture. Enfin, la table *LevelClass* contient le nom des classes du projet ainsi que la valeur des métriques calculées pour chacune des classes puis la table *ArchitectureMark* contient quant à elle la valeur finale de la qualité de l'architecture.

Fig. 4-7 - Diagramme de classes de notre base de données

4.3.4.2 Cardinalités et associations

Afin de pouvoir expliquer les différentes cardinalités et associations, penchons nous sur un exemple précis, le cas des relations entre les tables *PointofView*, *PointofViewFactorLink* et *FactorValue* (voir figure 4-8).

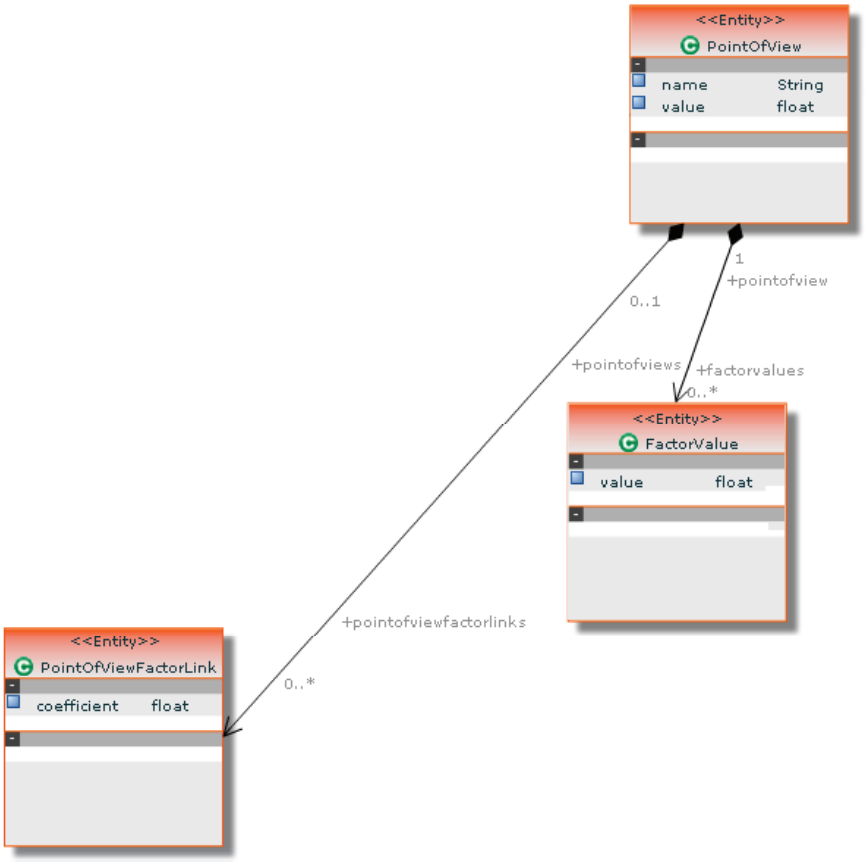

Fig. 4-8 - Relations entre les tables PointofView, PointofViewFactorLink et FactorValue

L'association entre les deux tables *PointofView* et *PointofViewFactorLink* puis celle entre les tables *PointofView* et *FactorValue* est appelée composite car les tables enfants (*PointofViewFactorLink* et *FactorValue*) et leurs enregistrements ne peuvent pas exister sans la table parent *PointofView* (à distinguer d'une association partagée

où les éléments enfants sont indépendants). La relation entre ces trois tables, consiste à dire que pour calculer la valeur (*value*) d'un certain point de vue de nom "*name*"; nous établissons la combinaison d'un enregistrement de la table *FactorValue* (l'attribut *value* représente la valeur d'un facteur et correspond à l'agrégation des valeurs pondérées des critères sous ce facteur) et d'un enregistrement de la classe *PointofViewFactorLink ;* l'attribut *coefficient* permettant de pondérer la valeur obtenue du facteur. Un enregistrement *PointofViewFactorLink* pointe vers zéro (uniquement pour s'éviter les erreurs lors de la création décalée des enregistrements) ou un seul point de vue et est associé à un enregistrement *FactorValue* qui pointe vers le même point de vue pour former une combinaison **unique** ; d'où les cardinalités de 1 pointant vers la classe PointofView. En ce qui concerne l'autre extrémité des associations (0..*), un point de vue peut bien entendu contenir plusieurs combinaisons de coefficients et valeurs de facteurs.

4.3.4.3 Base de données

4.3.4.3.1 Initialisation des tables de notre base de données

Ce diagramme de classes a été conçu à l'aide de l'outil BeoModeler, propriété de BeOtic. Cet outil assez puissant permet de créer très facilement un diagramme de classes qui est ensuite exporté et traduit sous forme textuelle (format .xmi pour XML Metadata Interchange Format).

Ce document contient la définition de toutes les tables et permet de créer automatiquement le squelette de notre base de données. Nous utilisons l'outil phpMyAdmin auquel nous soumettons en entrée le fichier XMI généré par l'outil BeoModeler et ainsi, notre base de données (tables, attributs puis que relations entre tables) est automatiquement créée (voir figure 4-9).

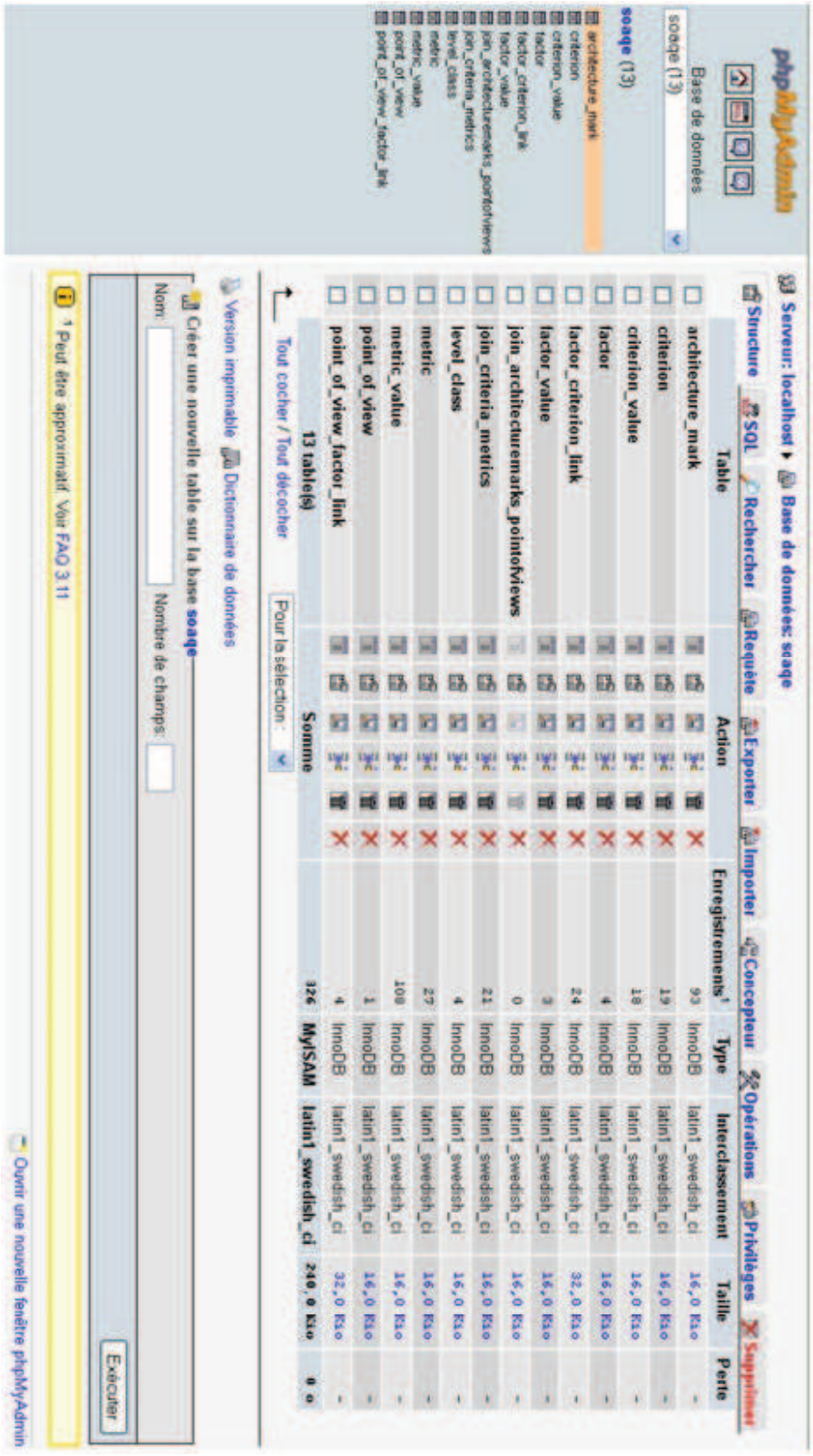

Fig. 4-9 - Schéma de la base de données

4.3.4.3.2 Alimentation des tables, enregistrement des métriques.

Pour alimenter les premières tables que nous caractérisons de socle pour notre base de données, nous avons implémenté un algorithme qui permet de parcourir le fichier "classes.xml" présenté dans la section 4.2.2 à l'aide de la technologie dom4j[32] ; une API qui permet d'analyser des sources XML :

```
SAXReader reader = new SAXReader();
            //loading the XML document from a file
Document document = reader.read(new
File(TestBase.class.getResource("classes.xml").getFile().replace("%20", " ")));
```

Ensuite, nous enregistrons la nouvelle métrique si elle n'existe pas déjà dans la base de données et nous lui affectons la valeur que nous lisons ; à contrario, si la métrique existe déjà, nous mettons à jour sa valeur avec la nouvelle valeur que nous lisons. Nous sauvegardons également la classe à laquelle la métrique appartient si cette classe n'existe pas déjà dans la base de données ; si elle existe, nous lui affectons la nouvelle valeur de la métrique lue.

4.3.5 Interface de l'outil

Notre base de données étant configurée et alimentée, nous pouvions ensuite commencer à concevoir le côté client de l'application. Pour notre version beta, nous avons choisi pour notre outil, une interface sobre et épurée où le code couleur est plutôt simple (voir figure 4-10) :

La couleur du fond de l'application est le noir et celle-ci est composée de deux parties majeures :

- La bannière, sur fond noir, contient le titre de l'application en blanc et les boutons de configuration. La couleur du fond des boutons est un dégradé de trois couleurs (noir, gris, blanc) et la couleur du texte est le blanc.

[32] http://dom4j.sourceforge.net/

- Le contenu où se concentre l'essentiel de l'application est sur fond blanc avec un menu vertical sur fond orange à gauche regroupant les différents onglets de l'application qui apparaît lorsque la souris le survole.

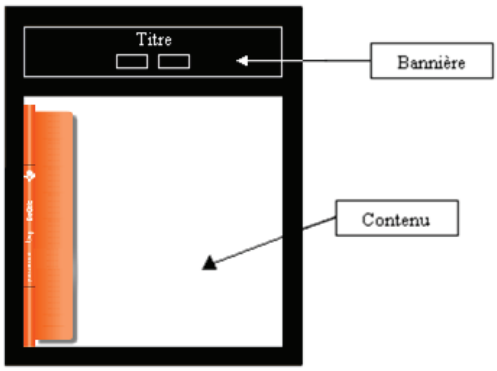

Fig. 4-10 - Interface de l'outil

L'outil permet de réaliser deux fonctionnalités principales :

1. **La visualisation des données** de l'architecture qui a été soumise à l'outil sous formes textuelle et graphique.
2. **L'évaluation de la qualité logicielle** de l'architecture qui a été soumise à l'outil.

4.4 La visualisation des données

Pour la première étape de notre application, nous avons implémenté un cubestack (parallélépipède rectangle dynamique de six faces dont quatre visibles en trois dimensions) pour une visualisation avancée du résultat et une ergonomie améliorée. Dans cette optique, l'utilisateur peut observer le comportement de l'ensemble des métriques contenues dans la base de données sous formes diverses et variées ; en effet, chacune des quatre faces visibles du cubestack nous présente une façon différente de visualiser les résultats de l'analyse (*scatterPlot*, *courbes*, *radar* et *datagrid*) pour une étude optimisée des résultats. Nous naviguons d'une face à

l'autre à l'aide d'une animation très ergonomique faisant pivoter le cube en direct. Nous accédons à chacune des faces du cube à l'aide d'un menu graphique sur fond orange contenant les onglets de chacun des modules (voir l'extrême gauche de la figure 4. 11).

1. Un premier module consiste à afficher, sous forme textuelle, dans une *datagrid* (grille de données) l'ensemble des valeurs des métriques retrouvées à partir de la base de données MySQL (face de droite du cubestack de la figure 4-11). Ces valeurs sont affichées en fonction des classes du code source. Ce module a été mis en place de manière à permettre à l'utilisateur de comparer simplement les métriques désirées pour l'évaluation avant le lancement effectif de celle-ci.

2. Un autre module *scatterPlot* (voir la face gauche du cubestack de la figure 4-11) est composé de trois axes. Chacun des axes est dynamique et contient la liste entière des métriques que l'on a sauvegardées en base. L'intérêt majeur est d'avoir d'un coup d'œil la visualisation de la comparaison (position et grosseur de la boule) de trois métriques qualité que l'on a sélectionnées parmi ceux de la liste.

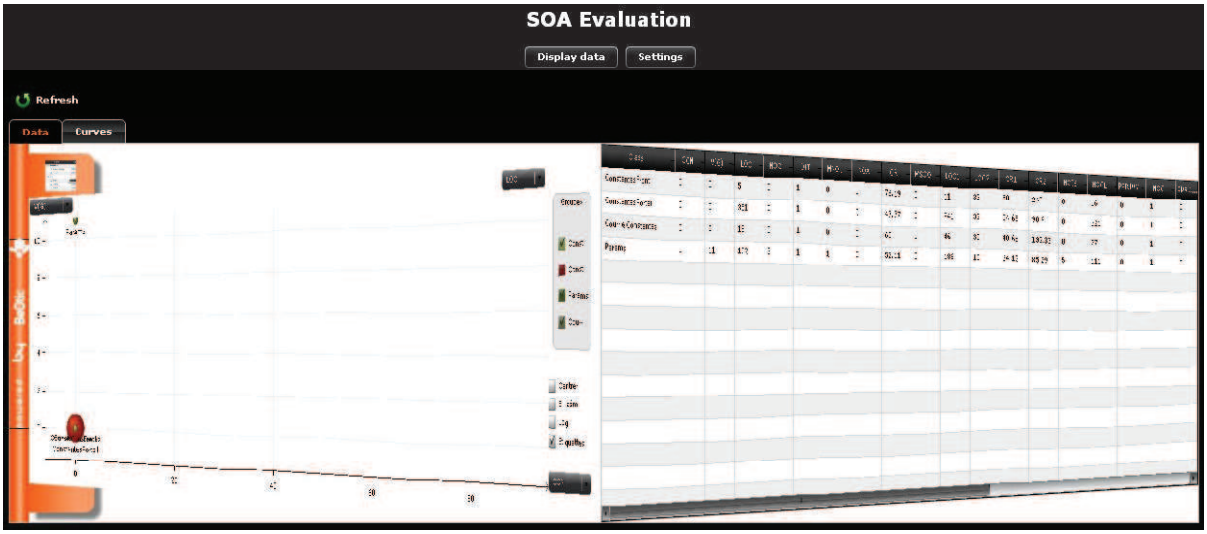

Fig. 4-11 - Capture d'écran de l'animation dynamique du passage du module *datagrid* (à droite) au module s*catterPlot (à gauche).*

3. Nous avons également implémenté le module *courbe*, où l'utilisateur peut visualiser le comportement des valeurs des métriques sélectionnées (nous pouvons sélectionner autant de métriques que l'on souhaite) en fonction de chacune des classes de l'architecture à travers des courbes colorées (voir le schéma 4-12). *Pour une simplification du schéma et pour éviter que les intitulés se superposent, nous avons ici affiché en abscisse l'ID des classes et non leur nom entier qui est très long car contenant leur chemin absolu.*

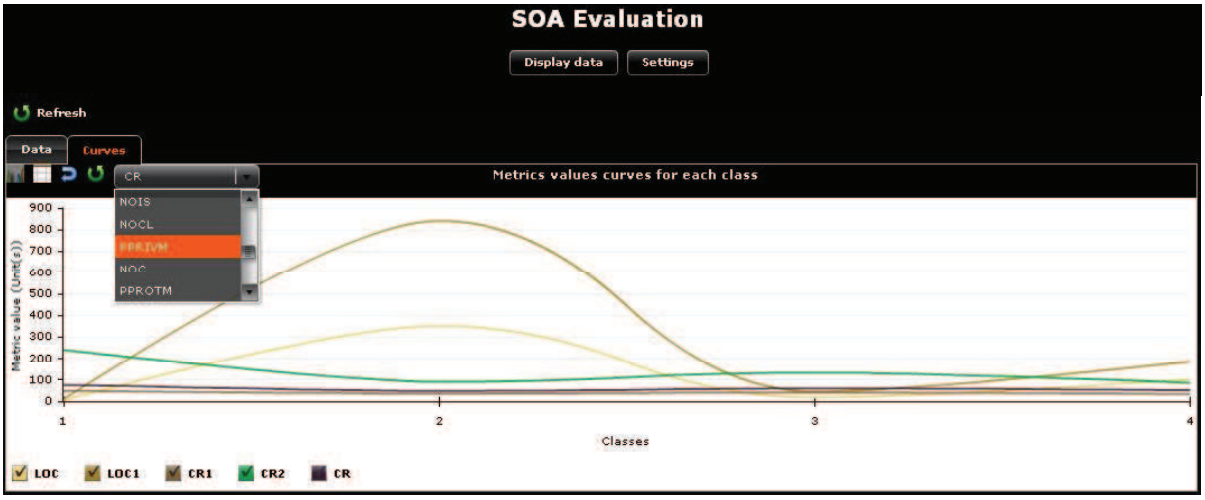

Fig. 4-12 - Module “Courbes” de l’outil SOAQE.

1. Et enfin, nous avons également mis en place le module "*radar*" qui nous affiche le comportement de quatre métriques choisies et comparées entre les classes du projet. Les classes à droite de la figure 4-9 sont différenciées par des couleurs et les métriques occupent les quatre extrémités des axes du radar. Sur la figure 4-13, les métriques sont identifiées par leur ID car c’est une version béta : l’axe d’extrémité "2" correspond à la métrique LOC, la métrique dont l’ID est "19" est CR, la métrique "7" est LOC1 et enfin la "20" représente la métrique V(G). A titre d’exemple, nous voyons que la classe "fr.company.vp.web.framework.constante.CourrielConstantes" représentée en rouge sur le radar a, en son sein, très peu de lignes de codes effectives. Nous pouvons également distinguer clairement sur la gauche le panneau permettant de passer d’une face à une autre du cubestack.

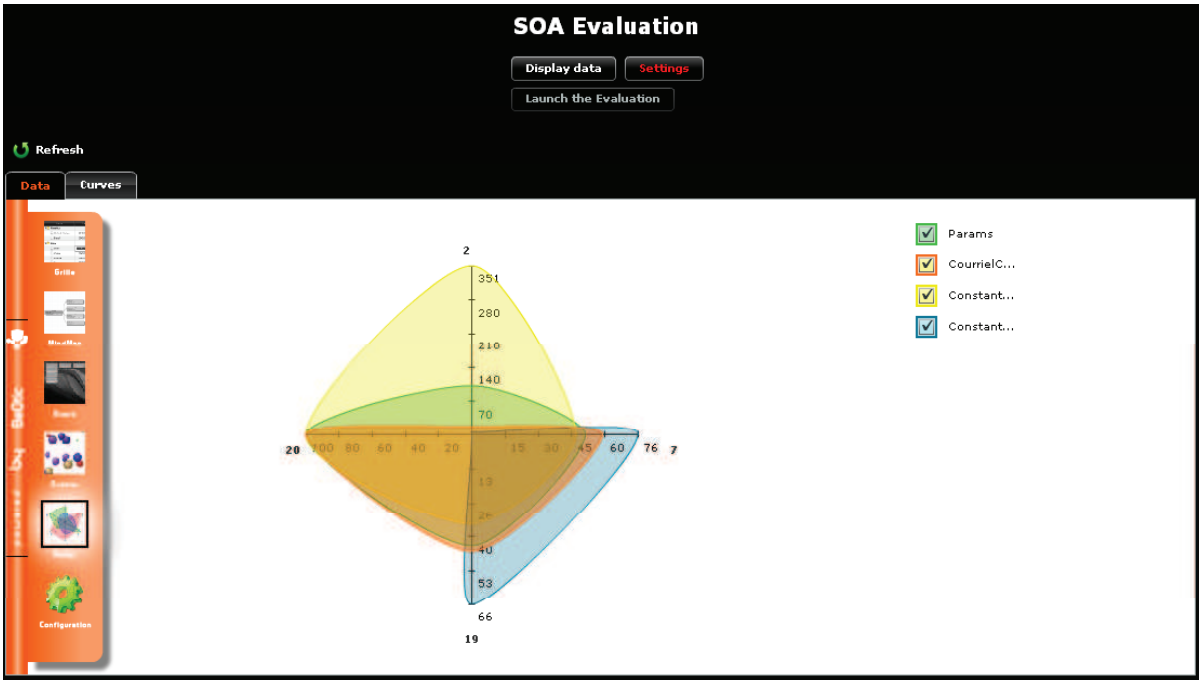

Fig. 4-13 - Module “Radar“ de l’outil SOAQE

Ces différents procédés d'affichage des données ont été mis en place pour obtenir une vision améliorée, très complète et avancée des résultats provenant de la base de données car d'expérience nous avons remarqué que des résultats affichés sous forme textuelle uniquement peuvent parfois perdre l'utilisateur. Avant de passer à l’évaluation de l’architecture qu’il a soumise, l’utilisateur a dorénavant en sa possession l’ensemble des données en jeu pour l’évaluation.

4.5 L’évaluation de l’architecture

Avant de lancer l'évaluation de l'architecture soumise, l'utilisateur peut configurer la structure de l’arbre de la partie de l'architecture étant évaluée (l’arbre est organisé en points de vue, facteurs, critères et métriques avec une notion de hiérarchisation et les données parviennent de la base de données). La structure de l’arborescence est

définie à l'aide d'un panneau ayant la forme d'une "datagrid" où est d'abord affichée à l'utilisateur une arborescence "par défaut" correspondant à la déclinaison la plus complète de l'architecture pour le point de vue architectural que nos précédents travaux ont identifiés [HkO10]. Cette arborescence est accompagnée de ses coefficients de pondération que nous avons également déterminés suite à nos travaux de recherche; ceci étant, afin que l'utilisateur puisse avoir une totale liberté d'actions pendant la phase d'évaluation, nous lui avons offert, à travers cette datagrid éditable, l'opportunité de toujours pouvoir :

(i) Modifier les attributs sélectionnés dans l'arborescence qu'on lui propose par défaut.

(ii) Ajouter de nouveaux attributs.

(iii) Supprimer des attributs existants.

(iv) Modifier les coefficients de pondération affichés par défaut

Il a été conclu dans les travaux précédents [BOV12c] (et comme nous l'avons indiqué dans la section 2) que seuls les facteurs qualité et les critères qualité doivent être pondérés car ces derniers n'ont pas la même importance dans l'architecture en fonction du point de vue considéré alors que les métriques définissent systématiquement leur critère de la même façon.

Le schéma 4-14 est un aperçu du panneau de commande.

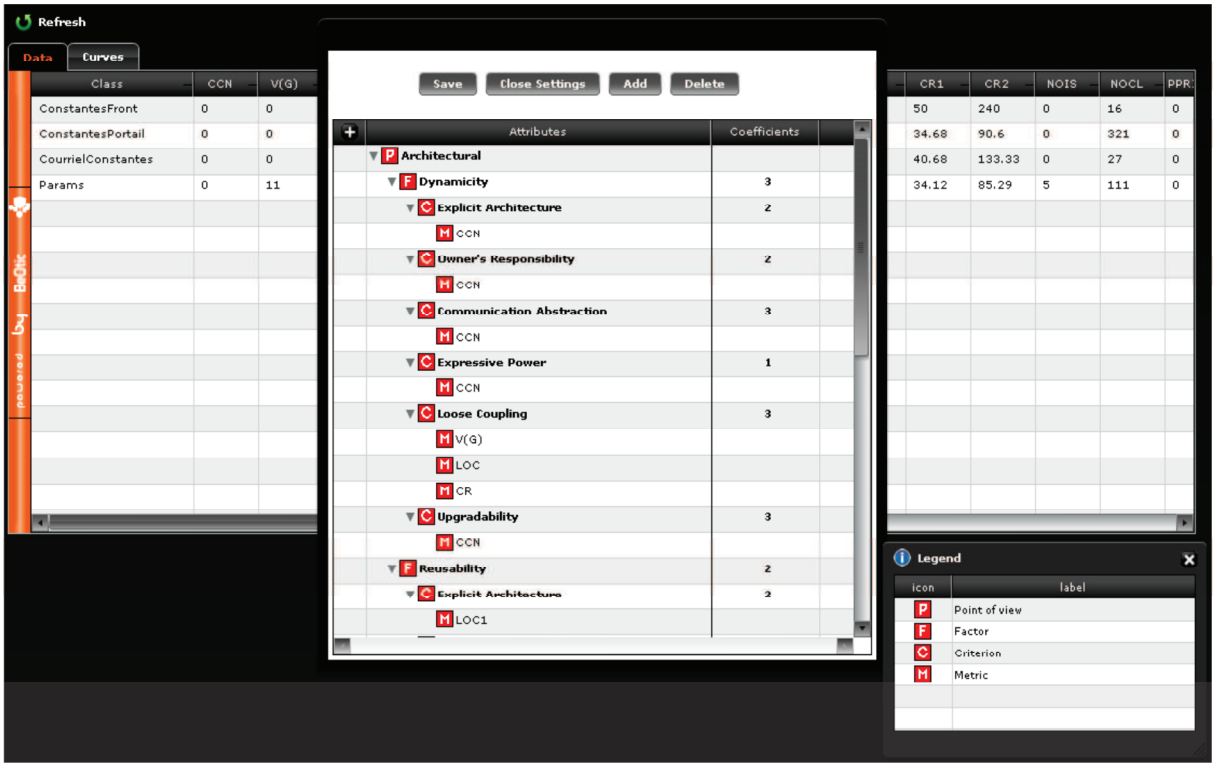

Fig. 4-14 - Control Panel

Après avoir modifié le panneau à sa guise afin d'y inscrire l'arborescence souhaitée, l'utilisateur clique sur le bouton "sauvegarder" et ainsi cette nouvelle arborescence est immédiatement mise à jour dans la base de données MySQL pour la prochaine étape de l'application : l'évaluation. Chaque niveau de l'arborescence est mis à jour dans sa table respective (inscription des nouveaux points de vue, facteurs, critères et métriques qualité dans la base de données). Les nouveaux coefficients sont également sauvegardés et une table de jointure des coefficients avec les attributs qualité respectivement correspondants est également mise à jour.

Plus concrètement, voici le mécanisme technique permettant de mener à bien ces différentes opérations, à savoir enregistrer la nouvelle structure de l'architecture ainsi que les différents coefficients.

L'application lit coté client la datagrid du panneau de configuration ; et envoie au serveur tous les points de vues (et leur contenu) qui existent dans la datagrid :

```
stock=new ArrayCollection();
```

```
            for each (var lOcc:Object in
mVisual.mTree.dataProvider.source.source){
                    if (lOcc.coeff==0){stock.addItem(lOcc);}}
                    if (stock!=null){mBeoCommand.proxyData.submitCoef(stock);}
```

Cette collection de points de vue soumise au serveur est ensuite parcourue et si le point de vue existe déjà en base, nous le mettons à jour ainsi que son contenu, s'il n'existe pas nous le créons et nous créons également son contenu (facteurs, critères, métriques).
Il s'agit d'une opération de configuration de l'architecture où intervient le parcours de la collection de points de vue, la mise à jour ou la création, en fonction de la situation et un travail établi au niveau du cran du dessous, les facteurs. En effet, un autre parcours en interne est réalisé afin de vérifier si les facteurs saisis dans le panneau de configuration existent ou non. S'ils existent, nous mettons à jour leurs données, y compris leur coefficient ; sinon, nous créons un nouveau facteur. Nous effectuons un travail similaire pour les niveaux du dessous (les critères) en revanche, les métriques ne sont pas créées car nous disposons d'une liste finie issue de l'outil BeoMetrics ; elles sont seulement affectées ou désaffectées.

4.5.1 Le résultat de l'évaluation

Ces opérations étant terminées, le panneau se ferme et un nouveau bouton "Lancer l'évaluation" apparaît. Cette nouvelle opération consiste à obtenir une valeur finie, exprimée en pourcentages, représentant la qualité de l'architecture soumise. Ce nombre correspond au résultat de l'agrégation de tous les attributs de la structure arborescente.

4.5.1.1 Traduction des valeurs des métriques.

Avant de fournir cette note globale, il existe plusieurs travaux qui sont réalisés côté serveur. Dans un premier temps, nous devions d'abord convertir chacune des valeurs des métriques lues en base, en pourcentages afin de qu'elles soient toutes dans la même unité dans le but de les combiner et les agréger. Pour ce, nous nous sommes basés sur une analyse profonde des travaux validés par la communauté de l'ingénierie logicielle étant liés à la définition de seuils de métriques [HGW11, AYV10, Kan03, RHS02, RH98, Hen96, BBM96,]. Il existe également un tableau très

complet sur le site McCabe [33] qui nous a beaucoup aidés : il liste les différentes métriques (et pas seulement celles de la suite de McCabe) ainsi que leurs seuils.

Il a par exemple été ainsi établi, qu'au-delà de la valeur "75" et en dessous de la valeur "5" pour la métrique CR1, la note de 0% lui est forcément attribuée car d'après la littérature, il ne faut pas qu'il y ait plus de 75% et moins de 5% de commentaires par classe du projet ; 100% lui étant attribué si il existe entre 20 et 40% de commentaires dans la classe.

En ce qui concerne la métrique V(G), si la valeur obtenue dépasse "30", cela témoigne d'une complexité de code trop importante et la note de 0% lui est attribuée tandis qu'une valeur inférieure à "10" lui permettait d'obtenir une note optimale de 100%. Nous avons défini que le mécanisme de traduction des valeurs en pourcentages des métriques est proportionnellement dégressif : 0% pour "30", 100% pour "10", donc 50% pour "20" etc.

Nous avons travaillé de la même manière pour l'ensemble des métriques que nous avons considéré avant de pouvoir les manier à notre guise.

Nous obtenons ainsi la valeur de chaque critère en identifiant la moyenne en pourcentages de toutes les métriques le constituant. Nous pondérons la valeur de chaque critère qualité obtenu, à l'aide de son coefficient de pondération correspondant. La moyenne des valeurs de tous les critères qualité pondérés de la même branche définira la valeur de leur facteur qualité correspondant et la combinaison des valeurs de tous les facteurs qualité également pondérés avec leurs coefficients respectifs déterminera la valeur du point de vue les englobant. Et finalement, la note globale de la qualité de l'architecture soumise correspondra à la moyenne des points de vue qualité constituant la dite architecture.

Pour une compréhension améliorée, voici schématiquement, sur la figure 4-15 ce que l'application effectue comme opérations sur un exemple d'architecture envisageable ; en partant

---

[33] http://www. Mccabe.com/pdf/McCabe IQ Metrics.pdf

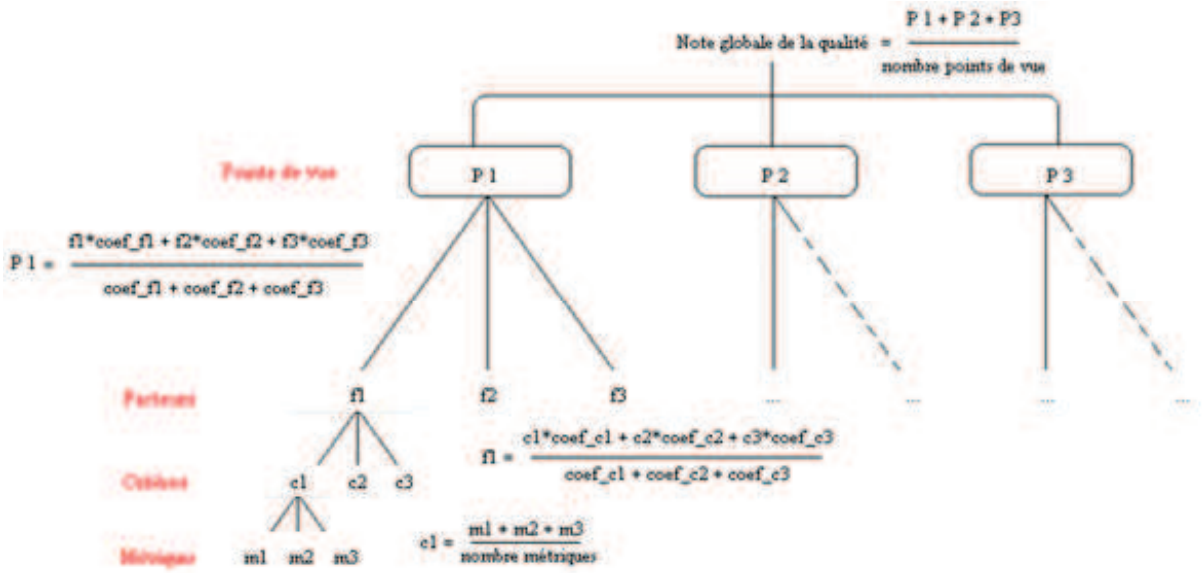

Fig. 4-15 - Arbre de calcul des notes

Au niveau du code, coté serveur, un clic sur le bouton "Lancer l'évaluation" appelle une fonction qui prend en entrée, comme pour la sauvegarde de la configuration, la liste des points de vue ainsi que leur contenu ayant été saisie sur la datagrid lors de la configuration de l'évaluation. Le bout de code ci-après est décisif car il retranscrit la quintessence de l'opération d'évaluation où nous pouvons observer les différentes étapes précédemment citées ; à savoir le calcul des moyennes des métriques déterminant chaque critère :

```
lSommedesmetric=lSommedesmetric/criterion.getChildren().length;
lUpCriVal.setValue(lSommedesmetric);
```

Puis nous réalisons la moyenne des critères pondérés pour déterminer chaque facteur :

```
lSommefactorvalue+=lUpCriVal.getValue()*lUpFacCri.getCoefficient();
lSommecritcoef+=lUpFacCri.getCoefficient();
lSommefactorvalue=lSommefactorvalue/lSommecritcoef;
lUpFacVal.setValue(lSommefactorvalue);
```

Ensuite nous établissons la moyenne des facteurs, chacun pondéré avec son coefficient respectif pour déterminer leur point de vue correspondant :

```
lSommepov+=lUpFacVal.getValue()*lUpPovFac.getCoefficient();
lSommefactcoef+=lUpPovFac.getCoefficient();
```

```
lSommepov=lSommepov/lSommefactcoef;
lUpPov.setValue(lSommepov);
```

Et finalement nous déterminons la valeur générique de la qualité en faisant la moyenne des points de vue de l'architecture :

```
lSommearchi+=lUpPov.getValue();
lSommearchi=lSommearchi/pValues.length;
```

Nota Bene : Ce bout de code est évidemment inscrit dans plusieurs structure itératives "for" afin de boucler autant de fois qu'il existe d'attributs qualité dans l'architecture.

4.5.1.2 Application de l'outil.

Comme stipulé dans la section 3.3.1; nos travaux pendant la thèse nous ont permis d'identifier précisément la composition (en métriques) du critère "couplage lâche" comme étant une combinaison des métriques RFC, CBO, LCOM, MPC, Fan In, Ce, Ca et V(G).

Nous n'avions en notre possession pour la phase d'expérimentation uniquement la valeur des métriques RFC, CBO, MPC et V(G) et n'avions pas encore la main sur le code de BeoMetrics pour y considérer et ajouter les métriques restantes pour définir entièrement le critère "couplage lâche". Nous travaillerons très prochainement sur cet aspect afin de compléter définitivement ce critère.

Afin d'obtenir un résultat correct en considérant les attributs que nous avions en notre possession, nous avons attribué des valeurs neutres à tous les attributs qualité de l'architecture que nous n'étions pas en mesure d'évaluer, tout en gardant, comme le montre la figure 4-16 les coefficients que l'on a défini après la phase d'étude de l'état de l'art dans le but de garder les bons rapports de proportionnalité. Nous en avons conclu pour l'étude que nous avons menée sur ce projet et ses 568 classes, et avec les informations précédemment considérées que la note de l'architecture était de **48%**.

Cette valeur finale *lSommearchi* est exprimée en pourcentages, puis est affichée à l'utilisateur comme étant le résultat final de l'évaluation.

La figure 4-16 permet uniquement de voir une partie du résultat de l'évaluation; en effet nous n'y avons retranscrit que sa partie textuelle (le résultat en pourcentages). Parce que le rendu des résultats n'est pas uniquement textuel, la société BeOtic nous a demandé de ne pas révéler le moindre aperçu des rendus graphiques de l'évaluation pour éviter toute fuite potentielle car c'est spécialement à ce niveau que notre outil permet une innovation et une réponse améliorée à la problématique de l'évaluation logicielle de la qualité.

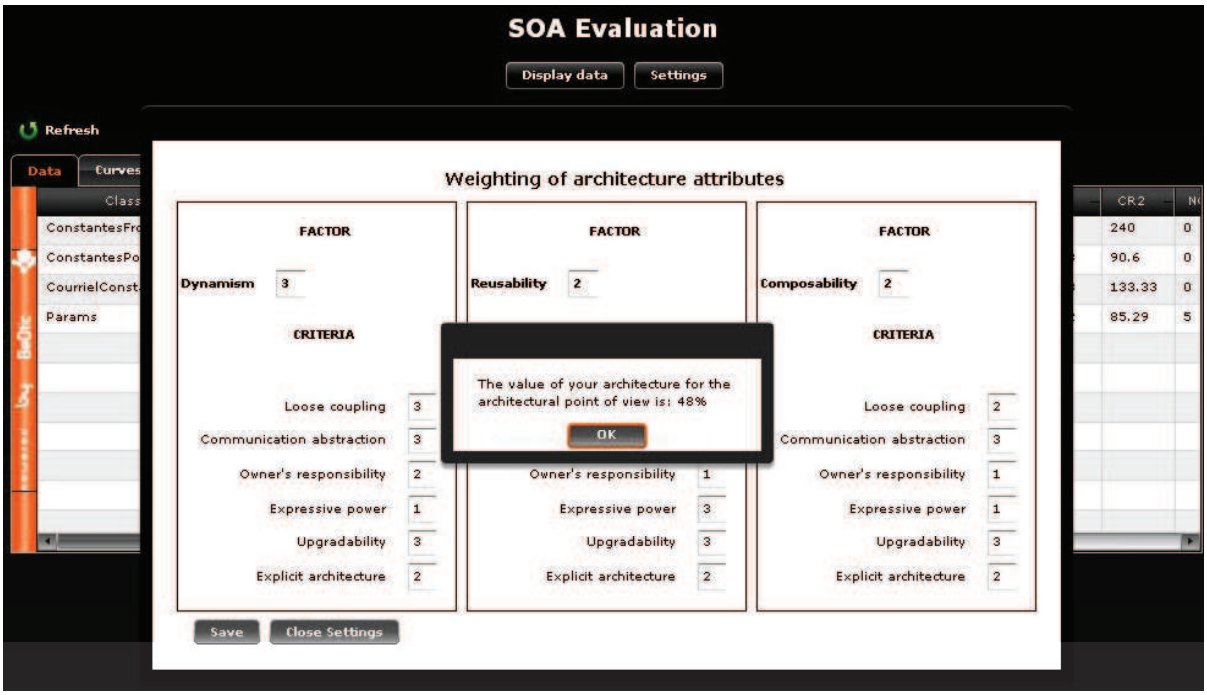

Fig. 4-16 - Résultat de l'évaluation sous forme textuelle uniquement

En ce qui concerne le reste des opérations de l'application ; une nouvelle fenêtre s'ouvre après avoir cliqué sur le bouton « OK » de la figure 4-16. Cette nouvelle fenêtre nous expose les résultats de l'évaluation sous diverses formes graphiques.

4.5 Conclusion

En guise de conclusion de cette section du manuscrit que nous avons dédiée à la phase d'implémentation de notre outil SOAQE à travers son aspect réalisationnel (architecture technique et technologies employées) puis fonctionnel (utilisation d'un jeu de données pour valider chacune des deux fonctionnalités de l'outil SOAQE), nous pouvons en déduire que les résultats sont positifs dans la mesure où ils nous permettent de dresser un premier bilan sur l'architecture logicielle mise en place au sein de l'organisation. En effet, une valeur de 48% en guise de résultat de l'outil est un premier élément qui nous permet de dire que le traitement du couplage lâche au sein de l'organisation est moyennement bon et qu'il peut encore être nettement amélioré. Bien entendu, de nouveaux travaux permettant de perfectionner l'outil sont en cours et c'est ce qui rend notre sujet de recherche passionnant dans la mesure où nous savons précisément les points qui restent perfectibles pour l'obtention d'un outil totalement fonctionnel et applicable. Ces perspectives futures sont dressées dans la section "conclusion et perspectives" qui suit.

Conclusion et Perspectives

L'étude des architectures logicielles et des paradigmes architecturaux, nous a conduits à nous pencher sur une problématique récurrente dans le domaine du génie logiciel : la mesure de la qualité d'un système logiciel.

Puisque SOA implique la notion complexe de connectivité entre plusieurs systèmes, entités commerciales et technologiques ; quelques compromis concernant l'architecture doivent être envisagés ; et parce que les décisions au sujet des SOA tendent à être pervasives, et, par conséquent, à avoir un impact important sur la société, la mise en place d'une évaluation de la qualité de l'architecture pendant la vie du logiciel est particulièrement cruciale.

Pendant l'évaluation des architectures logicielles, nous pesons la pertinence de chacune des problématiques associées à la phase de conception après avoir évalué l'importance de chaque exigence liée aux attributs qualité. Les résultats obtenus avec les méthodes existantes (ATAM, SAAM) lors de l'évaluation des architectures logicielles sont souvent très différents et aucune de celles-ci ne mène l'évaluation de

façon pertinente [CKK01]. Nous connaissons les causes de ce problème : la plupart des méthodes d'analyse et d'évaluation de la qualité des systèmes logiciels sont effectuées à partir du code source. D'un point de vue architectural, ces techniques peuvent être caractérisées comme étant de bas niveau, et peuvent ne pas correspondre aux projets basés sur de nouvelles architectures complexes.

Notre thèse s'inscrit totalement au cœur du sujet dans la mesure où nous avons tenté de répondre à cette problématique de l'évaluation logicielle, chère à la communauté du génie logiciel.

Cette conclusion revient sur un bilan des travaux étudiés, puis introduit les limitations de nos contributions s'ouvrant sur autant de perspectives.

Bilan

Durant notre thèse, nous avons d'abord dressé une classification des travaux existants de mesure de la qualité logicielle suivant différents critères tels que les objectifs de l'approche, le modèle de qualité utilisé ou les informations utilisées.

L'évaluation de la qualité d'une architecture orientée service, qu'elle soit conduite aussi bien sur une architecture en cours de développement ou sur une architecture existante, concerne des aspects qualitatifs et quantitatifs, la prévision de la charge associée aux évolutions et les limites théoriques d'une architecture donnée. Nos travaux nous ont permis d'identifier les forces des architectures logicielles dans ce domaine, mais nous avons aussi pu mettre en évidence les inconvénients de l'intervention des experts pour diriger un tel processus.

Si nous avons montré que l'expertise des architectes, permettait une évaluation précoce de la qualité d'un système ; cette intervention humaine a un impact direct sur le degré d'automatisation de l'évaluation de la qualité d'une architecture et sur son caractère reproductible. Ce point essentiel a dirigé nos recherches et nos travaux dans la mesure où nous voulions à tout prix limiter l'intervention humaine pour que

notre prototype soit semi automatisé. Nous avons également effectué un travail d'analyse des différents modèles de qualité ayant fait leur apparition ces dernières décennies. C'est ainsi que nous avons montré les avantages en termes de généricité et d'adaptation du modèle ayant servi de socle à nos travaux; le modèle de McCall [McC70] ayant conduit à la norme ISO/IEC 25010:2011 (anciennement ISO-9126-1) [SAA03].

A partir de celui-ci, nous avons donc conçu le modèle SOAQE (accompagné de sa méthode et de son outil) permettant de partitionner une architecture orientée service en une combinaison d'attributs qualité afin d'en évaluer la qualité.

Le processus général est basé sur deux étapes principales :

(i) La division de l'architecture en quatre niveaux d'attributs (points de vue, facteurs, critères et métriques qualité).

(ii) Le calcul de la note correspondant à la qualité de l'architecture.

Notre proposition offre une nouvelle manière de traiter la problématique de l'évaluation des architectures SOA. Afin d'obtenir un modèle et un outil qui permettent d'évaluer précisément la qualité d'une architecture orientée service, il est essentiel de pouvoir diviser l'architecture entière en une composition de plusieurs attributs organisés autour d'une arborescence prédéfinie. La finalité de notre travail fut de concevoir un cadre conceptuel et, in fine, un prototype semi-automatisé (basé sur les méthodes existantes, telles qu'ATAM ou SAAM) qui pouvait mesurer avec une valeur précise en guise de résultat la qualité globale de l'architecture orientée service soumise à notre outil.

Nous voulions fournir à l'utilisateur des résultats "moins abstraits" que ceux que les méthodes actuelles proposent aujourd'hui. Le concept de qualité demeurant relatif, nous visons les secteurs exigeant une attention particulière en s'adressant directement aux diverses équipes de développement en charge des fonctions concernées. Le prototype en question permet de réaliser des gains considérables en

termes de temps et d'argent contrairement à l'ensemble des travaux existants visant l'évaluation de la qualité d'une SOA [BCR94, CKK01, MW08, CDR03]

Le modèle sur lequel l'outil est basé a été déjà validé par la communauté de l'ingénierie logicielle [BOV12b] et permet d'obtenir des résultats d'évaluation de la qualité SOA réels, précis et immédiats. Cet outil qui a également été présenté à la communauté scientifique [BOV13] a été implémenté pour éviter les défaillances de projets majeurs. En effet, nous pouvons maintenant savoir s'il semble raisonnable pour une organisation de bouleverser son architecture existante et d'opter pour la technologie SOA ou si la configuration mise en place est celle qui convient le mieux. C'est d'ailleurs exactement sur ce point que la société BeOtic a éprouvé un réel intérêt pour le projet car cette société est spécialisée dans l'audit de services informatique et propose une large panoplie d'outils logiciels à ses clients. Néanmoins, nous avons travaillé sur ce projet en tant qu'architectes et le travail pour le point de vue architectural n'est pas complètement terminé car il y existe toujours des critères qui n'ont pas été décomposés en agrégations de métriques connues (nous y reviendrons plus loin dans le chapitre des perspectives futures). Ainsi même si l'outil fonctionne correctement et que les résultats obtenus sont justes et convaincants, il est encore possible d'apporter de nouveaux éléments au travail actuel. C'est pourquoi nous avons choisi de laisser l'utilisateur libre de modifier les arborescences proposées par défaut pour pouvoir incorporer au travail existant les nouveaux résultats de recherche que les travaux futurs permettront d'obtenir. Nous avons conçu la première fois un travail plutôt limité mais quand le prototype a considérablement évolué, nous avons ajouté de nouvelles fonctionnalités pour que l'outil soit le plus configurable possible pour l'utilisateur.

Dans cette optique, de nouvelles procédures concernent l'étude profonde de nouveaux critères pour le point de vue architectural. Une autre partie des perspectives concerne la recherche sur de nouveaux points de vue ; nous avons déjà commencé quelques recherches sur l'arborescence du point de vue "business".

Perspectives

Les perspectives des travaux futurs sont en rapport direct avec les limitations actuelles de nos propositions. Celles-ci peuvent donc être organisées autour de nos trois contributions principales : Le modèle prévisionnel de qualité, la méthode d'évaluation d'une architecture SOA et l'outil SOAQE. Pour une compréhension plus claire de la suite des travaux à entreprendre, nous distinguons deux vues englobant l'ensemble de nos contributions, à savoir, l'aspect conceptuel et l'aspect "*réalisationnel*".

Aspect conceptuel

Les perspectives qui se dégagent du travail conceptuel sont classées en deux catégories suivant l'importance du travail exploratoire effectué. Nous distinguons ainsi les perspectives à court terme et les perspectives plus lointaines.

Perspectives à court terme

En ce qui concerne les travaux d'ordre conceptuel à envisager à court terme, nous devons dans un premier temps compléter dans BeoMetrics la définition du critère "couplage lâche" en considérant les métriques restantes n'étant pas encore calculées ; à savoir, LCOM, Fan In, Ce et Ca mais cette opération n'est ni longue, ni compliquée, dans la mesure où dans la littérature il existe déjà le moyen de calculer ces métriques ; il suffit simplement de l'incorporer à BeoMetrics, l'outil très évolutif que BeOtic a conçu. Ensuite, il parait logique, après avoir étudié en profondeur le critère qualité "couplage lâche", qu'il faille en second lieu se concentrer essentiellement sur l'identification des métriques des cinq autres critères qualité que l'on a identifiés pour chacun des facteurs qualité (dynamicité, réutilisabilité, composabilité) sous la branche du point de vue qualité architectural. A savoir

l'évolutivité, l'abstraction de communication, la responsabilité propriétaire, l'architecture explicite et le "pouvoir expressif (entourés en bleu sur la figure 6-1).

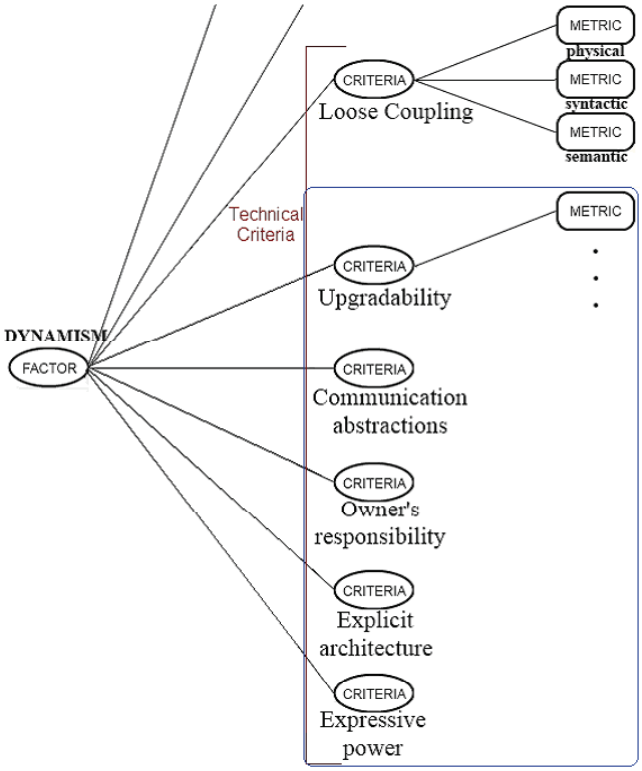

Fig. 5-1 - Attributs qualité sous le facteur "dynamicité"

Ainsi, lorsque nous connaîtrons l'architecture complète sous le point de vue architectural, à savoir connaître, et pouvoir calculer l'ensemble des métriques qualité de tous les critères qualité recensés, il sera alors possible de déterminer, comme nous l'avons fait pour le critère qualité "couplage lâche" la valeur de chaque critère de notre architecture, et donc chaque facteur pour exprimer une valeur finie de la qualité de l'ensemble du point de vue qualité architectural de l'architecture SOA soumise à notre outil.

Perspectives à long terme

Un travail que je classifierais à envisager à long terme consisterait à identifier l'ensemble des points de vues qualité restants d'une architecture SOA tels que le

point de vue qualité business par exemple, et à planifier pour chacun d'entre eux, un travail identique à celui qui a été réalisé pour le point de vue architectural durant notre thèse (entourés en bleu sur la figure 6-2).

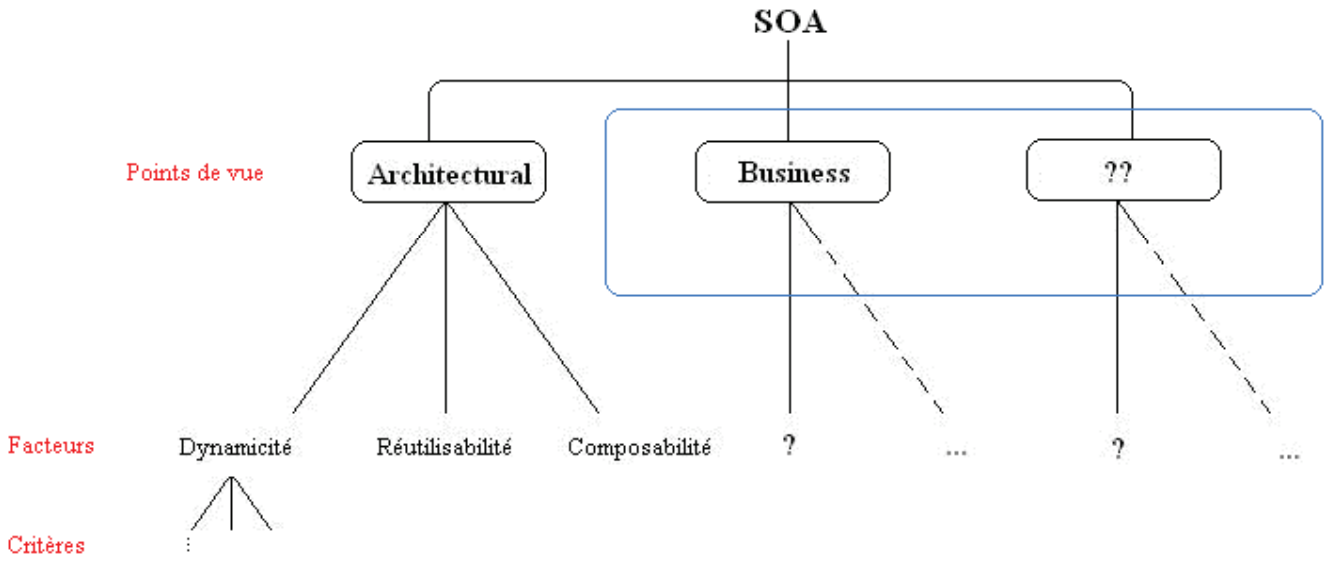

Fig. 5-2 - Points de vue qualité à explorer pour une SOA

Ceci afin de pouvoir dégager pour chacun une valeur de leur qualité. Ainsi, lorsque nous disposerons définitivement de la valeur finie de chacun des points de vue qualité de notre architecture SOA, nous pourrons établir la pondération puis l'agrégation de toutes les valeurs obtenues afin de dégager une note générique et définitive de la qualité de l'ensemble de l'architecture orientée service.

Aspect "*réalisationnel*"

De la même manière que pour l'aspect conceptuel, nous pouvons également distinguer pour les travaux d'ordre "*réalisationnel*", leurs perspectives à court terme et celles plus lointaines.

Perspectives à court terme

Nous envisageons avec BeOtic de consacrer prochainement beaucoup de temps à la finition du prototype que nous avons conçu pendant notre thèse ceci afin de le commercialiser in fine. Pour ce faire, il y a des travaux qui doivent être rapidement entrepris, notamment ceux qui concernent la visualisation graphique du résultat de l'évaluation. Dans l'état actuel des choses nous obtenons un résultat sous forme textuel (valeur en pourcentage de la qualité) et graphique (ScatterPlot). La visualisation sous forme de ScatterPlot est très intéressante dans la mesure où elle nous permet d'observer d'un coup d'œil, après évaluation, chacun des départements de l'architecture représenté par sa boule, accompagné de sa "note" ; ceci afin de rapidement identifier quels sont les secteurs de l'architecture faisant défaut et ceux qui se portent bien.

Nos travaux imminents concernent directement une représentation graphique poussée du résultat de l'évaluation afin qu'elle puisse évoluer vers une finalité innovante. Par soucis de confidentialité, nous ne pouvons pas décrire dans les détails l'aspect final du contenu graphique car c'est là où notre application tire son originalité.

Perspectives à long terme

Les travaux d'aspect "*réalisationnel*" à long terme sont fortement corrélés à ceux d'ordre conceptuel. Le travail qui a été conçu permet à l'outil d'être très facilement évolutif et adapté aux nouveaux travaux liés aux attributs qualité. Au niveau de l'implémentation, nous avions anticipé les évolutions futures liées à notre recherche scientifique et à l'identification de nouvelles métriques définissant les critères qualité que l'on connaît déjà. Il sera ainsi possible de rapidement les intégrer à l'outil et les traiter. En effet, le pont entre la base de données et l'outil est complètement dynamique ; ainsi, lorsque l'utilisateur entrera les nouvelles métriques dans la base de données, celles-ci seront automatiquement affichées sur l'interface de l'outil qui est déjà très modulable.

De plus, malgré que nous nous soyons, pendant la thèse, clairement identifiés comme étant des intervenants au niveau architectural exclusivement, et qu'ainsi nous avons décidé de travailler uniquement avec des attributs qualité que l'on qualifie de « technique », nous avons tout de même pensé à concevoir un outil générique capable de considérer de nouveaux points de vue qualité. Au niveau de la base de données, nous pourrons donc facilement ajouter ces points de vue qualité qui seront également automatiquement affichés sur l'interface de l'outil. Ce que nous pouvons imaginer à long terme concerne des petits travaux secondaires liés à la maintenance de l'outil et à son esthétisme et ses fonctionnalités, en fonction des requêtes des clients.

Liste Des Publications De Nos Travaux

Conférences internationales avec actes et comité de lecture

R.Belkhatir, M. Oussalah et A. Viguier - **A Method and a Tool for Evaluating the Quality of an SOA**. Software Engineering Research and Practice (SERP) 2013, Las Vegas**.**

R.Belkhatir, M. Oussalah et A. Viguier. SOAQE - **Service Oriented Architecture Quality Evaluation**. The International Conferences on Evaluation of Novel Approaches to Software Engineering (ENASE) 2012, Wrocław: 199-202.

R.Belkhatir, M. Oussalah et A. Viguier - **SOAs factors, criteria and metrics**. Software Engineering Research and Practice (SERP) 2012, Las Vegas.

R.Belkhatir, M. Oussalah et A. Viguier - **A Model Introducing SOAs Quality Attributes Decomposition**. The International Conference on Software Engineering and Knowledge Engineering (SEKE) 2012, San Francisco: 324-327.

Journaux internationaux

R.Belkhatir, M. Oussalah et A. Viguier - **An industrial case study on SOA quality evaluation.** International Journal on Engineering Applications (IREA) 2013 ISSN 2281-2881.

Liste Des Abbréviations

CIFRE : Conventions Industrielles de Formation par la REcherche
LINA : Laboratoire Informatique de Nantes Atlantique
AeLoS : Architectures et Logiciels Sûrs
OOA : Object Oriented Architecture
CBA : *Component Based Architecture*
SOA : *Service Oriented Architecture*
SOAQE : *Service Oriented Architecture Quality Evaluation*
ATAM : *Architecture tradeoff analysis method*
SAAM : *Software architecture analysis method*
GSA : *General Services Administration*
ARID : *Active Reviews for Intermediate Designs*
SACAM : *Software Architecture Comparison Analysis Methods*
OO : Orienté objet
ADL : *Architecture Description Language*

XML : *Extensible Markup Language*
GAO : *Government Accounting Office*

Bibliographie

[ACK+02] Alencar, P.S.C., Cowan, D.D., Kunz, T. and Lucena, C.J.P. A formal architectural design patterns-based approach to software understanding. In *Proceedings of Fourth workshop on program comprehension*, pages 154-163, 1996. issn 1092-8138.

[AL11a] April, A. and Laporte, C.Y. L'assurance qualité logicielle: Tome 1, Concepts de base, Lavoisier, 2011. isbn 978-2746231474.

[AL11b] April, A. and Laporte, C.Y. L'assurance qualité logicielle 2 : processus de support, Lavoisier, 2011. isbn 978-2746232228.

[Als03] Alshayeb, M. and Li, W. An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes, *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pages 1043-1049, Nov. 2003.

[ARK05] Alghamdi, J.S., Rufai, R.A and Khan, S.M. OOMeter: A Software Quality Assurance Tool. *Proceedings of the Ninth European*

Conference on Software Maintenance and Reengineering, CSMR'05, 1534-5351/05, 2005.

[AYV10] Alves, T.L., Ypma, C. and Visser, J. Deriving metric thresholds from benchmark data, *IEEE International Conference on Software Maintenance* (ICSM), pages 1-10, 2010. issn 1063-6773.

[Bac03] Bachmann, F. Deriving architectural tactics: A step toward methodical architectural design, 2003, Technical report, Carnegie Mellon University. Software Engineering Institute. asin B0006S82M4

[Bai12] Bailet, T. Architecture logicielle, pour une approche organisationnelle, fonctionnelle et technique, Eni, 2012. isbn 978-2746073852

[BBK+78] Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., McLeod, G. and Merritt, M., Characteristics of Software Quality, North Holland, 1978.

[BBL76] Boehm, B.W., Brown, J.R. and Lipow, M., Quantitative evaluation of software quality, *International Conference on Software Engineering, Proceedings of the 2nd international conference on Software engineering*, pages 592-605, 1976.

[BBM96] Basili, V.R., Briand, L.C. and Melo, W.L., A Validation of Object-Oriented Design Metrics as Quality Indicators, *IEEE Transactions on Software engineering, Piscataway*, NJ, USA, pages 751-761, October 1996. issn 0098-5589.

[BCK03] Bass, L., Clements, P. and Kazman, R., Software architecture in practice. Addison Wesley, 560 pages, Avril 2003. isbn 978-0321815736.

[BCR94] Basili, V.R., Caldiera, G. and Dieter Rombach, H., The Goal Question Metric Approach. *Chapter in Encyclopedia of Software Engineering*, Wiley, 1994

[BDH+98] Broy, M., Deimel, A., Henn, J., Koskimies, K., Plasil, F., Pomberger, G., Pree, W., Stal, M. and Szyperski, C.A., What characterizes a (software) component? *Software - Concepts and Tools* 19(1): 49-56, 1998.

[BD02] Bansiya, J. and Davis, C.G., A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[BEL+03] Barbacci, M., Ellison, R., Lattanze, A., Stafford, J., Weinstock, C. and Wood, W., Quality Attribute Workshops, QAWs. *Third Edition (CMU/SEI-2003-TR-016). Software Engineering Institute, Carnegie Mellon University*, 2003. http ://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6687.

[Ber05] Berger, M., Data Access Object Pattern, 2005.

[BKM07] Bianco, P., Kotermanski, R. and Merson, O., Evaluating a service oriented architecture. CMU/SEI-2007-TR-015, Carnegie Mellon University, Software Engineering Institute, 2007.

[BLB+04] Bengtsson, P., Lassing, N., Bosch, J. and Vliet, H., Architecture-level modifiability analysis (ALMA), *Journal of Systems and Software*, Volume 69, Issues 1-2, pages 129-147, January 2004.

[BOV12a] Belkhatir, R., Oussalah, M. and Viguier, A., SOAQE - Service Oriented Architecture Quality Evaluation. *The International Conferences on Evaluation of Novel Approaches to Software Engineering, ENASE*, 2012, Wrocław.

[BOV12b] Belkhatir, R., Oussalah, M. and Viguier, A., A Model Introducing SOAs Quality Attributes Decomposition. *The International Conference on Software Engineering and Knowledge Engineering, SEKE*, 2012, San Francisco.

[BOV12c] Belkhatir, R., Oussalah, M. and Viguier, A., SOAs factors, criteria and metrics. *Software Engineering Research and Practice, SERP*, 2012, Las Vegas.

[BOV13] Belkhatir, R., Oussalah, M. and Viguier, A., A Method and a Tool for Evaluating the Quality of an SOA. *Software Engineering Research and Practice, SERP,* 2013, Las Vegas.

[CBA02] CBAM: Cost Benefit Analysis Method. 2002. http ://www.sei.cmu.edu/ata/products_services/cbam.html

[CCL06] Crnkovic, I., Chaudron, M. and Larsson, S., Component based development process and component lifecycle. *International Conference on Software Engineering Advances, ICSEA*, 2006.

[CDK98] Chidamber, S.R., Darcy, D.P. and Kemerer, C.F., Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on software Engineering*, vol. 24, No 8, pp 629 – 639, August 1998.

[CDR03] Cotroneo, D., Di Flora, C. and Russo, S., Improving Dependability of Service Oriented Architectures for Pervasive Computing, *Proceedings of The Eighth IEEE International Workshop on Object-Oriented Real-Time*, 2003.

[CF06] Carvallo, J.P., and Franch, X., Extending the ISO/IEC 9126-1 Quality Model with Non-Technical Factors for COTS Components Selection, *In Proceedings of the 2006 international workshop on Software quality, WoSQ '06*. ACM, New York, pages 9-14. 2006.

[CGB+02] Clements, P., Garlan, G., Bass, L., Stafford, J., Nord, R., Ivers, J. and Little, R., Documenting Software Architectures: Views and Beyond. *Pearson Education*, 2002.

[CK91] Chidamber, S.R. and Kemerer, C.F., Towards a Metrics Suite for Object Oriented Design, OOPSLA '91. *Conference Proceedings, Special Issue of SIGPLAN Notices*, Vol. 26, No. 11, pages 197–211, November 1991.

[CK94] Chidamber, S.R. and Kemerer, C.F., A Metrics Suite for Object Design, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pages 476–493, June 1994.

[CKK01] Clements, P., Kazman, R. and Klein, M., Evaluating Software Architectures: Methods and case studied, *Addison-Wesley Professional*, 2001.

[CKK02] Clements, P., Kazman R. and Klein, M., Evaluating software architectures: methods and case studies. *Quality Attribute Workshop, QAW, Addison-Wesley*, 2002.

[CL00] Cavarero, J.L. and Lecat, R., La conception orientée objet, évidence ou fatalité, Ellipses, 2000.

[Cro79] Crosby, P. B., Quality is free: the art of making quality certain, *McGraw-Hill*, 1979.

[CS08] Carlson, J. and Stadmeyer, K., FLEX, AMF 3 and BlazeDS: An Assessment, *Black Hat USA 2008*, 2008.

[CS09] Chung, L., Sampaio do Prado Leite, J.C., On non-functional requirements in software engineering. *Conceptual modelling: foundations and applications, pages 363-379,* 2009. isbn 978-3-642-02462-7.

[Dem86] Demarco, T., Controlling software projects: management, measurement and estimates, *Prentice Hall*, 296 pages, 1986.

[Dem88] Deming, W.E., Out of the crisis: quality, productivity and competitive position, *Cambridge University Press*, 1988.

[DN02a] Dobrica, L. and Niemelä, E., A survey on software architecture analysis methods. *IEEE Transaction on Software Engineering*, pages 638-653, July 2002.

[DN02b] Dobrica, L. and Niemelä, E., Software Reuse: Methods, Techniques, and Tools in Software Architecture Quality Analysis Methods, pages 15-27, 2002.

[Dol02] Dolan, T.J., Ph.D. Thesis, Architecture Assessment of Information-System Families, *Department of Technology Management, Eindhoven University of Technology*, February 2002.

[Dro95] Dromey, R.G., A model for software product quality, *IEEE Transactions on Software Engineering*, no. 2, pages 146-163, 1995.

[Dro96] Dromey, R.G., Concerning the Chimera (software quality), *IEEE Transactions on Software Engineering*, no. 1, pages 33-43, 1996.

[Els84] Elshoff, J.L., Characteristic program complexity measures. *In Proceedings of the 7th international conference on Software engineering*, *ICSE '84,* pages 288-293, 1984.

[Fab07] Fabresse, L., Ph.D. Thesis, Du découplage à l'assemblage non-anticipé de composants : conception et mise en œuvre du langage à composants, 2007.

[Fei83] Feigenbaum, A.V., Total quality control, *McGraw-Hill*, 1983.

[FV03] Faust, D. and C. Verhoef., Software product line migration and deployment. *Software Practice and Experience, John Wiley & Sons, Ltd*, 33(10): 933-955, 2003.

[Gar84] Garvin, D.A., What does 'Product Quality' really mean?, *Sloan Management Review*, no. 1, pages 25-43,1984.

[GHJ+94] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, *Addison-Wesley*, 1994.

[GKT05] Gröne, B., Knöpfel, A. and Tabeling, P. Component vs. Component: Why We Need More Than One Definition. *12th IEEE International Conference on the Engineering of Computer-Based Systems, ECBS,* pages 550-552, 2005.

[Gov77] Government Accounting Office. Government-wide guidelines and management assistance center needed to improve ADP systems development, 1977. http ://www.gao.gov.

[Gor06] Gorman, J. Object Oriented Design Principles & metrics, 2006.

[Gra92] Grady, R.B. Practical software metrics for project management and process improvement, *Prentice Hall*, 270 pages, 1992.

[GS94] Garlan, D. and Shaw, M. An introduction to software architecture, CMU/SEI-94-TR-21, ESC-TR-94-21, 1994.

[Hab94] Habrias, H. La mesure du logiciel, Teknea, 1994.

[Hal77] Halstead, M.H. Elements of Software Science (Operating and Programming Systems Series). *Elsevier Science Inc.*, 1977.

[Hen96] Henderson-Sellers, B. Object-Oriented Metrics, measures of Complexity, *Prentice Hall*, 1996.

[HGW11] Herbold, S., Grabowski, J. and Waack, S. Calculation and optimization of thresholds for sets of software metrics, *Empirical Software Engineering*, Volume 16, Issue 6, pp 812-841, December 2011.

[HH01] Hoyer, R.W. and Hoyer, B.B.Y. What is quality?, *Quality Progress*, no. 7, pages 52-62, 2001.

[HkO10] Hock-koon, A. and Oussalah, M. Expliciting a composite service by a metamodeling approach. *In Research Challenges in Information Science RCIS*, 2010.

[HR96] Hyatt, L.E. and Rosenberg, L.H. A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality, *European Space Agency Software Assurance Symposium and the 8th Annual Software Technology Conference*, 1996.

[Hsu01] Hsu, J.Y. Computer architecture, software aspects, coding, and hardware, *CRC Press*, 2001.

[IHO02] Ionita, M., Hammer, D. and Obbink, H. Scenario-based software architecture evaluation methods: an overview. *In Workshop on Methods and Techniques for Software Architecture Review and Assessment at the International Conference on Software Engineering*, 2002.

[Ish85] Ishikawa, K. What is total quality control? : The Japanese way, *Prentice-Hall*, 1985.

[Iso01] ISO, International Organization for Standardization, ISO 9126-1:2001, Software engineering – Product quality, Part 1: Quality model, 2001.

[Jon11] Jones, L. An Architecture-Centric Approach for Acquiring Software-Reliant Systems, *DTIC Document*, 2011.

[Jur88] Juran, J.M. Juran's Quality Control Handbook, *McGraw-Hill*, 1988.

[Kaz01] Kazman, R. Software Architecture-Guest Editors' Introduction. *International Journal of Software Engineering and Knowledge Engineering* 11(4): 387, 2001.

[Kan03] Kan, S.H. Metrics and models in software Quality Engineering, *Addison-Wesley*, 2003.

[KBA+94] Kazman, R., Bass, L., Abowd, G. and Webb, M. SAAM: A Method for Analyzing the Properties Software Architectures, *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy*, pages 81-90, May 1994.

[KKB+98] Kazman, R., Klein, M., Barbacci, M., Lipson, H., Longstaff, T. and Carriere, J. The Architecture Tradeoff Analysis Method. *In Proceedings Fourth International Conference Engineering of Complex Computer Systems, ICECCS '98*, August 1998.

[KKC00] Kazman, R., Klein, M. and Clements, P. ATAM: Method for architecture evaluation: ATAM-Architecture Trade-off Analysis Method report, 2000. http ://www.sei.cmu.edu/ata/ata_method.html.

[KKL+01] Kabali, H., Keller, R.K., Lustmaan F. and Saint-Denis, G. Class cohesion as predictor of changeability: An Empirical study, *ACM Digital Library*, 2001.

[Kim08] Kim, H.K. Modeling of distributes systems with SOA and MDA, *International Journal of Computer Science, IAENG*, vol: 35; Issue: 4; Start page: 509, 2008. issn 1819-656X.

[KP96] Kitchenham, B. and Pfleeger, S.L. Software quality: the elusive target, *IEEE Software*, no. 1, pages 12-21, 1996.

[Lap09] Laplante, P. Requirements engineering for software and systems, *1st edition. CRC Press, Redmond*, 2009.

[Las02] Lassing, N. Ph.D. Thesis, Architecture-Level Modifiability Analysis, *Free University Amsterdam*, February 2002.

[Leg09] LeGoaer, O. Ph.D. Thesis, Styles d'évolution dans les architectures logicielles, 2009.

[Lév14] Lévy, N. Chapitre "Architecture et qualité de systèmes logiciels" dans "Architecture logicielles, principes techniques et outils", Lavoisier, 2014.

[LHB85] Lehman, M.M., Hünke, H. and Boehm, B.W. (Eds.): *Proceedings, 8th International Conference on Software Engineering, IEEE Computer Society*, August 1985. isbn 0-8186-0620-7.

[LMB07] Lero, O., Merson, P. and Bass, L. Quality attributes and service oriented architectures, *In Proceedings of the International Workshop on Systems Development in SOA Environments, SDSOA*, 2007.

[LRV99] Lassing, N., Rijsenbrij, D. and van Vliet, H. On Software Architecture Analysis of Flexibility, Complexity of Changes: Size Isn't Everything. *In Proceedings Second Nordic Software Architecture Workshop, NOSA '99*, pages 1103-1581, 1999.

[Mar00] Martin, C.R. Design Principles & design patters, *Objectmentor*, 2000.

[Mar02] Martin, C.R. Agile Software Development: Principles, Patterns, and Practices, *Pearson Education, Prentice Hall*, 2002.

[Maz06] Mazumder, S. SOA: A Perspective on Implementation Risks. *SETLabs Briefings publication,* October 2006.

[McC70] McCall, J.A. Economics of information and job search, *Quarterly Journal of Economics,* 84 (1): 113–126, 1970.

[Mol99] Molter, G. Integrating SAAM in Domain-Centric and Reuse-Based Development Processes, *In Proceedings of Second Nordic Workshop Software Architecture, NOSA '99*, pages 1103-1581, 1999.

[Mor02] Mortureux, Y. Preliminary risk analysis. Techniques de l'ingénieur. Sécurité et gestion des risques, 2002. SE2(SE4010):SE4010.1-SE4010.10.

[MRW77] McCall, J.A., Richards, P.K. and Walters, G.F. Factors in Software Quality, *National Technical Information Service*, no. vol. 1, 2 and 3, 1977.

[MW08] Magott, J. and Woda, M. Evaluation of SOA security metrics using attack graphs, *IEEE 2008*, pages 277-284, 2008.

[NBC+03] Nord, R., Barbacci, M., Clements, P., Kazman, R., Klein, M., O'Brien, L. and Tomayko, J. Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM) (CMU/SEI-2003-TN-038). *Software Engineering Institute, Carnegie Mellon University*, 2003. http ://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6557.

[Oa99] Oussalah, M. and all. Génie objet, analyse et conception de l'évolution, *Lavoisier*, 1999. isbn 2-7462-0029-5.

[Oa05] Oussalah, M. and all. Ingénierie des composants : Concepts, techniques et Outils. *Vuibert*, 2005. isbn 2711748367.

[Oa14] Oussalah, M. and all. Architectures logicielles principes techniques et outils, *Lavoisier*, 2014.

[OS08] Oussalah, M. and Smeda, A. COSABuilder : an Extensible Tool for Architectural Description, *3rd International Conference on Telecom Technology and Applications, ICTTA 2014*, pages 1–6, 2008.

[PB93] Parobeck, F. and Bonno, G. La qualité logicielle. Concepts de base et mise en œuvre, *Dunod*, 1993.

[PRF07] Perepletchikov, M., Ryan, C., Frampton, K. and Tari, Z. Coupling metrics for predicting maintainability in service-oriented designs, *The 23rd Australasian Software Engineering Conference, ASWEC*:329–340, 2007.

[Pri12] Printz, J. Architecture logicielle, concevoir des applications simples sures et adaptable 3ème édition, *Dunod*, 2012.

[PW92] Perry, D.E. and Wolf, A.L. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, pages 40-52, 1992. http://doi.acm.org/10.1145/141874.141884.

[Rat03] Rational Software Inc. RUP - Rational Unified Process, 2003. http :// www.rational.com.

[RG08] Roy, B. and Graham, T.C.N. Methods for evaluating software architecture: A survey. *School of Computing TR*, 545: 82, 2008.

[RH98] Rosenberg, L.H and Hyatt, L. Applying and Interpreting Object Oriented Metrics, *Software Assurance Technology Conference, Utah*, 1998.

[RHS02] Rodríguez, D., Harrison, R. and Satpathy, M. A Generic Model and Tool Support for Assessing and Improving Web Processes. *IEEE METRICS.* pages: 141-151, 2002.

[Rom02] Roman, S. Part V, Data Access Objects in Access database, design and programming, *O'Reilly Media*, 2002.

[SAA03] Suryn, W., Abran, A. and April, A. ISO/IEC SQuaRE: The second generation of standards for software product quality, *International Association of Science and Technology for Development, IASTED*, 2003.

[SBV03] Stoermer, C., Bachmann, F. and Verhoef, C. SACAM: The Software Architecture Comparison Analysis Method, *Software Engineering Institute* CMU/SEI-2003-TR-006, December 2003.

[SG96] Shaw, M. and Garlan, D. Software architecture: perspectives on an emerging discipline. *Upper Saddle River, N.J. Prentice Hall*, 1996.

[She31] Shewhart, W.A. Economic control of quality of manufactured product, *Van Nostrand*, 1931.

[Som11] Sommerville, I. Software Engineering (9th Edition), *Pearson/Addison-Wesley*, 773 pages, 2011.

[Sun02] Sun Microsystems, Core J2EE Patterns - Data Access Object, 2002, http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

[Sun12] Sun, W. An Implementation of Flex Access to Remote JAVA Object Based on the BlazeDS Framework, *Journal of Jining Normal University*, 2012.

[Swe06] Sweden, E. Service oriented architecture: An enabler of the agile enterprise in state government, Nascio Research Brief, 2006. http ://www.nascio.org.

[Szy02] Szyperski, C. Objectively: Components versus Web Services?, *Object-Oriented Programming, Lecture Notes in Computer Science, the European Conference on Object-Oriented Programming, ECOOP 2002*, Volume 2374, p 256, 2002.

[Tiw09] Tiwari, S. Professional BlazeDS: Creating Rich Internet Applications with Flex and Java, *Wiley Publishing*, 2009.

[Zha99] Zhao, F. A cooperative framework for process planning, *International Journal of Computer Integrated Manufacturing*, Volume 12, Issue 2, pages 168-178, 1999. DOI:10.1080/095119299130407.

Contribution to the automation and to the evaluation of open-system software architectures

Riad BELKHATIR

Abstract

These last years, service oriented architectures (SOA) drew the attention of the software engineering community so much their utility in terms of coupling improvements, reusability and productivity showed its mettle. Last SOA solutions tend to be more granular and "measurable", thus, one of the most exciting challenges of the last decades consists in being able to evaluate quantitatively the quality of a software architecture. That primarily makes it possible to control the various costs and to prevent possible risks. During this thesis, we essentially focused on a new semi automated method allowing the evaluation of the software quality of service oriented architectures. Our contributions are summarized in three axes. The first one presents a model of quality, inspired from the McCall model, which splits any service oriented architecture into a hierarchical tree, organized around several quality attributes. The second axis relates to a semi automated method called SOAQE stemming from this model and allowing the evaluation of service oriented architectures. Then, the third axis presents the SOAQE tool, based on the method of the same name, returning a combination of the software quality evaluation results in textual and graphic forms for a better understanding of the data.

Keywords: Software architecture paradigms, Service oriented architectures, Quality evaluation, quality factor, quality criterion, quality metric.

Contribution à l'automatisation et à l'évaluation des architectures logicielles ouvertes

Résumé

Ces dernières années, les Architectures Orientées Service (AOS) ont attiré l'attention de la communauté de l'ingénierie logicielle tant leur utilité en termes d'amélioration du couplage, de réutilisabilité et de productivité a fait ses preuves. Les dernières solutions AOS tendent à être plus granulaires et "mesurables", ainsi, un des challenges les plus excitants des dernières décennies consiste à être capable d'évaluer quantitativement la qualité d'une architecture. Cela permet essentiellement de contrôler les différents coûts ainsi que de prévenir d'éventuels risques. Durant cette thèse, nous nous sommes essentiellement focalisés sur une nouvelle méthode semi automatisée permettant l'évaluation de la qualité logicielle des architectures orientées service. Nos contributions se résument en trois axes. Le premier présente un modèle de qualité, inspiré du modèle de McCall, qui décompose une architecture orientée service en un arbre hiérarchisé de plusieurs attributs qualité. Le second concerne une méthode semi automatisée appelée SOAQE dérivant de ce modèle et permettant l'évaluation d'architectures orientées service. Tandis que le troisième axe présente l'outil SOAQE basé sur la méthode éponyme retournant une combinaison de résultats de l'évaluation de la qualité logicielle sous formes textuelle puis graphique pour une meilleure compréhension des données.

Mots-clés: Paradigmes d'architecture logicielle, Architectures orientées service, Évaluation de la qualité, Facteur qualité, Critère qualité, Métrique qualité.